CPBPV: a constraint-programming framework for bounded program verification

Hélène Collavizza, Michel Rueher, Pascal van Hentenryck

CPBPV: A Constraint-Programming Framework for Bounded Program Verification*

Hélène Collavizza[1], Michel Rueher[1], Pascal Van Hentenryck[2]

[1] University of Nice–Sophia Antipolis, I3S/CNRS, BP 145, 06903 Sophia Antipolis Cedex, France (helen@polytech.unice.fr, michel.rueher@gmail.com)
[2] Brown University, Box 1910, Providence, RI 02912 (pvh@cs.brown.edu)

Abstract. This paper studies how to verify the conformity of a program with its specification and proposes a novel constraint-programming framework for bounded program verification (CPBPV). The CPBPV framework uses constraint stores to represent both the specification and the program and explores execution paths of bounded length nondeterministically. The CPBPV framework detects non-conformities and provides counter examples when a path of bounded length that refutes some properties exists. The input program is partially correct under the boundness restrictions, if each constraint store so produced implies the post-condition. CPBPV does not explore spurious execution paths, as it incrementally prunes execution paths early by detecting that the constraint store is not consistent. CPBPV uses the rich language of constraint programming to express the constraint store. Finally, CPBPV is parameterized with a list of solvers which are tried in sequence, starting with the least expensive and less general. Experimental results often produce orders of magnitude improvements over earlier approaches, running times being often independent of the size of the variable domains. Moreover, CPBPV was able to detect subtle errors in some programs for which other frameworks based on bounded model checking have failed.

1 Introduction

This paper is concerned with software correctness, a critical issue in software engineering. It proposes a novel constraint-programming framework for bounded program verification (CPBPV). The goal is to verify the conformity of a program with its specification, that is, to demonstrate that the specification is a consequence of the program under the boundness restrictions. The key idea in CPBPV is to use constraint stores to represent both the specification and the program, and to non-deterministically explore execution paths of bounded length over these constraint stores. Non-determinism occurs when executing conditional or iterative instructions. The non-deterministic constraint-based symbolic execution incrementally refines the constraint store, which initially consists of the

* This work has been partially supported by the “CAVERN” ANR-07-SESUR-003 project and by the “TESTEC” ANR-07-TLOG 022-05 project.

precondition, by adding constraints coming from conditions and from assignments. CPBPV is a CP framework for *bounded* program verification, i.e., it assumes a bound on the program inputs (e.g., the array lengths and the variable values) and on the number of iterations for loops.[3] Boundedness guarantees that CPBPV terminates but it may induces incompleteness: Indeed, the CPBPV verifier is inconclusive if executable paths with a length greater than the specified bound exist. The input program is correct if each constraint store produced by the symbolic execution implies the post-condition and the loop unwinding assertion (terminating a loop early) does not fail. In particular, CPBPV can prove correctness of programs with a runtime bound, which is highly desirable for embedded applications.

It is important to emphasize that verifying the conformity between a program and its specification requires to check (explicitly or implicitly) all executables paths. This is not the case in some model-checking tools designed to detect violations of some specific property, e.g., safety or liveness properties. Relations between our CPBPV approach, model checking and bounded model checking are discussed in section 6.2.

The CPBPV framework has a number of fundamental benefits. First, contrary to earlier work using model checking or constraint programming [2, 16, 17], CPBPV does not use predicate abstraction or abstraction refinement techniques. As a consequence, it does not explore spurious execution paths, i.e., paths that do not correspond to actual executions over inputs satisfying the pre-condition. CPBPV incrementally prunes execution paths early by detecting that the constraint store is not consistent. Second, CPBPV uses the rich language of constraint programming to express the constraint store, including arbitrary logical and threshold combinations of constraints, the *element* constraint, and global/combinatorial constraints that express complex relationships on a set of variables. Finally, CPBPV is parametrized with a list of solvers which are tried in sequence, starting with the least expensive and least general.

The CPBPV framework has been evaluated experimentally on a series of benchmarks from program verification. Experimental results of our prototype often produce orders of magnitude improvements over earlier approaches, and indicate that the running times are often independent of the variable domains. Moreover, CPBPV is able to find subtle errors that some other verification frameworks could not detect.

The rest of the paper is organized as follows. Section 2 illustrates how CPBPV handles constraint stores on a motivating example. Section 3 formalizes the CPBPV framework for a small imperative programming language and Section 4 discusses the implementation issues. Section 5 presents experimental results on a number of verification problems, comparing our approach with state-of-the-art verification frameworks. Section 6 discusses related work in test generation, bounded program verification and software model checking. Section 7 summarizes the contributions and presents future research directions.

[3] For some benchmarks, we also add a bound on the size of the integers but this restriction is just required for efficiency reasons.

```
/*@ requires (\forall int i; i>=0 && i<t.length-1;t[i]<=t[i+1])
  @ ensures
  @   (\result != -1 ==> t[\result] == v) &&
  @   (\result == -1 ==> \forall int k; 0 <= k < t.length ; t[k] != v) @*/
1 static int binary_search(int[] t, int v) {
2    int l = 0;
3    int u = t.length-1;
4    while (l <= u) {
5      int m = (l + u) / 2;
6      if (t[m]==v)
7          return m;
8      if (t[m] > v)
9         u = m - 1;
10      else
11        l = m + 1;       // ERROR u = m - 1;
12   }
13   return -1;
14 }
```

Fig. 1. The Binary Search Program.

2 The Constraint-Programming Framework at Work

This section illustrates the CPBPV verifier on a motivating example, the binary search program. CPBPV uses java-like syntax and JML specifications[4] for pre- and post-conditions, appropriately enhanced to support the expressivity of constraint programming. Figure 1 depicts a binary search program to determine whether a value v is present in a sorted array t. (Note that `\result` in JML corresponds to the value returned by the program). To verify this program, our prototype implementation requires a bound on the length of array t, on its elements, and on v. We will verify its correctness for specific lengths and simply assume that the values are signed integers using a number of bits. We can also specify that the main loop is unwinded at most $\log(n) + 1$ times, which is the worst case complexity of the binary search algorithm for an array of length n, although this bound is not used by CPBPV to handle this example.

The initial constraint store of the CPBPV verifier, assuming an input array of length 8, is the precondition[5] $c_{pre} \equiv \forall 0 \leq i < 7 : t^0[i] \leq t^0[i+1]$ where t^0 is an array of constraint variables capturing the values stored in the input array t. The constraint variables are annotated with a version number as CPBPV performs an SSA-like[6] renamings [15] on the fly, since each assignment generates constraints possibly linking the old and the new values of the assigned variable. The assignments in lines 2–3 add the constraints $l^0 = 0 \wedge u^0 = 7$. CPBPV then considers the loop instruction. Since $l^0 \leq u^0$, the execution enters the loop body, adds the constraint $m^0 = (l^0 + u^0)/2$, which simplifies to $m^0 = 3$

[4] See `http://www.cs.ucf.edu/~leavens/JML/`

[5] We omit the domain constraints on the variables for simplicity.

[6] SSA : Single State Assignment

(3.5 is rounded down to the integer 3, since / is the division on integers), and considers the conditional statement on lines 6–7. The execution of the statement is nondeterministic: Indeed, both $t^0[3] = v^0$ and $t^0[3] \neq v^0$ are consistent with the constraint store, and thus the two alternatives, which give rise to two execution paths, must be explored. Note that these two alternatives correspond to actual execution paths in which $t[3]$ in the input is equal to, or different from, input v. The first alternative adds the constraint $t^0[3] = v^0$ to the store and executes line 7, which adds the constraint $result = m^0$. CPBPV has thus obtained a complete execution path p whose final constraint store c_p is:

$$c_{pre} \ \wedge \ l^0 = 0 \wedge u^0 = 7 \ \wedge \ m^0 = (l^0 + u^0)/2 \ \wedge \ t^0[m^0] = v^0 \ \wedge \ result = m^0$$

CPBPV then checks whether this store c_p implies the post-condition c_{post} by searching for a solution to $c_p \ \wedge \ \neg c_{post}$. This test fails, indicating that the execution path p, which captures the set of actual executions in which $t[3] = v$, satisfies the specification. CPBPV then explores the other alternatives to the conditional statement in line 6. It adds the constraint $t^0[m^0] \neq v^0$ and executes the conditional statement in line 8. Once again, this statement is nondeterministic. Its first alternative assumes that the test holds, generating the constraint $t^0[m^0] > v^0$ and executing the instruction in line 9. Since u is (re-)assigned, CPBPV creates a new variable u^1 and posts the constraint $u^1 = m^0 - 1 = 2$. The execution returns to line 4, where the test now reads $l^0 \leq u^1$, since CPBPV always uses the most recent version for each variable. Since the constraint store entails $l^0 \leq u^1$, the only extension to the current path consists in executing line 5, adding the constraint $m^1 = (l^0 + u^1)/2$, which actually simplifies to $m^1 = 1$. Another complete execution path is then obtained by executing lines 6 and 7.

Consider now a version of the program in which line 11 is replaced by `u = m-1`. To illustrate the CPBPV verifier, we specify partial execution paths by indicating which alternative is selected for each nondeterministic instruction. For instance, $\langle T_4, F_6, T_8, T_4, T_6 \rangle$ denotes the last execution path discussed above in which the true alternative is selected for the first execution of the instruction in line 4, the false alternative for the first execution of instruction 6, the true alternative for the first execution of instruction 8, the true alternative of the second execution of instruction 4, and the true alternative of the second execution of instruction 6. Consider the partial path $\langle T_4, F_6, F_8 \rangle$ and let us study how it can be extended. The partial path $\langle T_4, F_6, F_8, T_4, T_6 \rangle$ is not explored, since it produces a constraint store containing

$$c_{pre} \ \wedge \ t^0[3] \neq v^0 \ \wedge \ t^0[3] \leq v^0 \ \wedge \ t^0[1] = v^0$$

which is clearly inconsistent. Similarly, the path $\langle T_4, F_6, F_8, T_4, F_6, T_8 \rangle$ cannot be extended. The output of CPBPV on this incorrect program when executed on an array of length 8 (with integers coded with 8-bits to make it readable) produces, in 0.025 seconds, the counterexample:

$v^0 = -126 \ \wedge \ t^0 = [-128, -127, -126, -125, -124, -123, -122, -121] \ \wedge \ result = -1.$

This example highlights a few interesting benefits of CPBPV.

1. The verifier only considers paths that correspond to collections of actual inputs (abstracted by constraint stores) which satisfy the precondition. The resulting execution paths must all be explored since our goal is to prove the partial correctness of the program.
2. The performance of the verifier is independent of the integer representation on this application: it only requires a bound on the length of the array.
3. If an execution path is found which violates the specification, the verifier returns a counter-example for debugging the program.

Note that CBMC, a state-of-the-art bounded model checker, fails to verify this example on arrays of lengths greater than 32, whereas CPBPV is able to verify instances of length 256. ESC/Java2, a state-of-the-art static analysis tool, fails to verify this example unless loop invariants are provided (see the discussion in Section 5).

3 Formalization of the Framework

In this section, we first formalize the CPBPV verifier on a small abstract language using a small-step structural operational semantics. This semantics primarily specifies the execution paths over constraint stores explored by the verifier. Then we give the operational description of the verifier by presenting in more depth the verification algorithm that manages constraint stores.

3.1 Structural Operational Semantics

Syntax Figure 2 depicts the syntax of the programs and the constraints generated by the verifier. In the following, we use s, possibly subscripted, to denote elements of a syntactic entity S.

Renamings CPBPV creates variables and arrays of variables "on-the-fly" when they are needed. This process resembles a SSA[7] normalization but, does not introduce the join nodes, since the results of different execution paths are not merged. Similar renamings are used in model checking. The renaming uses mappings of type $V \cup A \rightarrow \mathcal{N}$ which maps variables and arrays into natural numbers denoting their current "version numbers". In the semantics, the version number is incremented each time a variable or an array element is assigned. We use $\sigma_\perp$ to denote the uniform mapping to zero (i.e., $\forall x \in V \cup A : \sigma_\perp(x) = 0$) and $\sigma[x/i]$ the mapping σ where x is now mapped to i, i.e., $\sigma[x/i](y) = if\, y = x \; then \; i \; else \; \sigma(y)$. These mappings are used by a polymorphic renaming function ρ to transform program expressions into constraints. For example, $\rho \; \sigma \; b_1 \oplus b_2 = (\rho \; \sigma \; b_1) \oplus (\rho \; \sigma \; b_2)$(where $\oplus \in \{\wedge, \vee, \Rightarrow\}$) is the rule used to transform a logical expression.

[7] SSA (static single assignment) form is an intermediate representation used in compiler dessign, in which every variable is assigned exactly once

L : *list of instructions*; I : *instructions*; B : *Boolean expressions*
E : *integer expressions*; A : *arrays*; V : *variables*

$L ::= I; L \mid \epsilon$
$I ::= A[E] \leftarrow E \mid V \leftarrow E \mid \mathit{if}\ B\ I \mid \mathit{while}\ B\ I \mid \mathit{assert}(B) \mid \mathit{enforce}(B) \mid \mathit{return}\ E \mid \{L\}$
$B ::= true \mid false \mid E > E \mid E \geq E \mid E = E \mid E \neq E \mid E \leq E \mid E < E$
$B ::= \neg B \mid B \wedge B \mid B \vee B \mid B \Rightarrow B$
$E ::= V \mid A[E] \mid E + E \mid E - E \mid E \times E \mid E/E \mid$

C : *constraints* E^+ : *solver expressions*
$V^+ = \{v^i \mid v \in V \ \&\ i \in \mathcal{N}\}$: *solver variables*
$A^+ = \{a^i \mid a \in A \ \&\ i \in \mathcal{N}\}$: *solver arrays*

$C ::= true \mid false \mid E^+ > E^+ \mid E^+ \geq E^+ \mid E^+ = E^+ \mid E^+ \neq E^+ \mid E^+ \leq E^+ \mid E^+ < E^+$
$C ::= \neg C \mid C \wedge C \mid C \vee C \mid C \Rightarrow C$
$E^+ ::= V \mid A[E^+] \mid E^+ + E^+ \mid E^+ - E^+ \mid E^+ \times E^+ \mid E^+/E^+ \mid$

Fig. 2. The Syntax of Programs and Constraints.

Configurations The CPBCV semantics mostly uses configurations of the type $\langle l, \sigma, c\rangle$, where l is the list of instructions to execute, σ is a version mapping, and c is the set of constraints generated so far. It also uses configurations of the form $\langle \top, \sigma, c\rangle$ to denote final states and configurations of the form $\langle \bot, \sigma, c\rangle$ to denote the violation of an assertion. The semantics is specified by rules of the form $\frac{\text{conditions}}{\gamma_1 \longmapsto \gamma_2}$ stating that configuration γ_1 can be rewritten into γ_2 when the conditions hold.

[S1] Conditional Instructions The conditional instruction *if* $b\ i$ considers two cases. If the constraint c_b associated with b is consistent with the constraint store, then the store is augmented with c_b and the body is executed. If the negation $\neg c_b$ is consistent with the store, then the constraint store is augmented with $\neg c_b$. Both rules may apply, since the store may represent some memory states satisfying the condition and some violating it.

$$\frac{c \wedge (\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{if}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle i\ ;\ l, \sigma, c \wedge (\rho\ \sigma\ b)\rangle} \quad \frac{c \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{if}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c \wedge \neg(\rho\ \sigma\ b)\rangle}$$

[S2] Iterative Instructions The while instruction *while* $b\ i$ also considers two cases. If the constraint c_b associated with b is consistent with the constraint store, then the constraint store is augmented with c_b, the body is executed, and the while instruction is reconsidered. If the negation $\neg c_b$ is consistent with the constraint store, then the constraint store is augmented with $\neg c_b$.

$$\frac{c \wedge (\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{while}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle i; \mathit{while}\ b\ i\ ;\ l, \sigma, c \wedge (\rho\ \sigma\ b)\rangle}$$

$$\frac{c \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{while}\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c \wedge \neg(\rho\ \sigma\ b)\rangle}$$

[S3] Scalar Assignments Scalar assignments create a new constraint variable for the program variable to be assigned and add a constraint specifying that the variable is equal to the right-hand side. A new renaming mapping is produced.

$$\frac{\sigma_2 = \sigma_1[v/\sigma_1(v)+1] \;\; \& \;\; c_2 \equiv (\rho\ \sigma_2\ v) = (\rho\ \sigma_1\ e)}{\langle v \leftarrow e\ ;\ l, \sigma_1, c_1\rangle \longmapsto \langle l, \sigma_2, c_1 \wedge c_2\rangle}$$

[S4] Assignments of Array Elements The assignment of an array element creates a new constraint array, add a constraint for the index being indexed and posts constraints specifying that all the new constraint variables in the array are equal to their earlier version, except for the element being indexed. Note that the index is an expression that may contain variables as well, giving rise to the well-known *element* constraint in constraint programming [35] in c_2.

$$\frac{\begin{array}{l}\sigma_2 = \sigma_1[a/\sigma_1(a)+1]\\ c_2 \equiv (\rho\ \sigma_2\ a)[\rho\ \sigma_1\ e_1] = (\rho\ \sigma_1\ e_2)\\ c_3 \equiv \forall i \in 0..a.length : (\rho\ \sigma_1\ e_1) \neq i \;\Rightarrow\; (\rho\ \sigma_2\ a)[i] = (\rho\ \sigma_1\ a)[i]\end{array}}{\langle a[e_1] \leftarrow e_2\ ; l, \sigma_1\ , c_1\rangle \longmapsto \langle l, \sigma_2, c_1 \wedge c_2 \wedge c_3\rangle}$$

The CPBCV semantics also features `assert` and `enforce` constructs which are necessary for modular composition.

[S5] Assert Statements An assert statement checks whether the assertion is implied by the control store in which case it proceeds normally. Otherwise, it terminates the execution with an error.

$$\frac{c \Rightarrow (\rho\ \sigma\ b)}{\langle assert\ b\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c\rangle} \qquad \frac{c \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle assert\ b\ ;\ l, \sigma, c\rangle \longmapsto \langle \bot, \sigma, c\rangle}$$

[S6] Enforce Statements An enforce statement adds a constraint to the constraint store if it is satisfiable.

$$\frac{c \wedge (\rho\ \sigma\ b) \text{ is satisfiable}}{\langle enforce\ b\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c \wedge (\rho\ \sigma\ b)\rangle}$$

[S7] Block Statements Block statements simply remove the braces.

$$\langle \{l_1\}\ ;\ l_2, \sigma, c\rangle \longmapsto \langle l_1 : l_2, \sigma, c\rangle$$

[S8] Return Statements A return statement simply constrains the *result* variable.

$$\frac{c_2 \equiv (\rho\ \sigma_1\ result) = (\rho\ \sigma_1\ e)}{\langle return\ e\ ;\ l, \sigma_1, c_1\rangle \longmapsto \langle \sigma_1, c_1 \wedge c_2\rangle}$$

[S9] Termination Termination also occurs when no instruction remains (empty list of instruction is denoted by ϵ).

$$\langle \epsilon, \sigma, c\rangle \longmapsto \langle \top, \sigma, c\rangle$$

The CPBPV Semantics Let $\mathcal{P}$ be the program $b_{pre}\ l\ b_{post}$ in which b_{pre} denotes the precondition, l is a list of instructions, and b_{post} the post-condition. Let $\overset{*}{\longmapsto}$ be the transitive closure of $\longmapsto$. The final states are specified by the set

$$SFN(\mathcal{P}) = \{\ \langle f, \sigma, c\rangle | \langle l, \sigma_\bot, \rho\ \sigma_\bot\ b_{pre}\rangle \overset{*}{\longmapsto} \langle f, \sigma, c\rangle\ \wedge\ f \in \{\bot, \top\}\ \}.$$

The program violates an assertion if the set

$$SFAE(\mathcal{P}) = \{\langle \bot, \sigma, c\rangle | \langle \bot, \sigma, c\rangle \in SFN(\mathcal{P})\}$$

is not empty. It violates its specification if the set

$$SFE(\mathcal{P}) = \{\langle \top, \sigma, c\rangle \in SFN(\mathcal{P})\ |\ c\ \wedge\ (\rho\ \sigma\ \neg b_{post}) \text{ is satisfiable}\}$$

is not empty. It is partially correct otherwise.

3.2 The Bounded Semantics

We now specialize the operational semantics to ensure termination. The only change is to limit the number of loop unfoldings to a bound B and to capture the fact that the verifier may be inconclusive (this is denoted by a configuration of the type $\langle ?, \sigma, c\rangle$). The semantic rules for iterative instructions now become:

[S2b] Iterative Instructions The while instruction *while b i* i first transformed into its bounded version $bwhile(B)\ b\ i$, where B is a limit on the number of loop unfoldings provided by the verifier:

$$\langle while\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle bwhile(B)\ b\ i\ ;\ l, \sigma, c\rangle$$

The bounded while instruction considers three cases. If the negation of the constraint c_b associated with b is consistent with the constraint store, then the constraint store is augmented with $\neg c_b$. If c_b is consistent with the constraint store and the maximal number of unfoldings is not reached, then the constraint store is augmented with c_b, the body is executed, and the bounded while instruction is reconsidered with one fewer possible unfolfings. Otherwise, if the maximal number of unfoldings is reached, the computation is inconclusive.

$$\frac{c \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle bwhile(n)\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle l, \sigma, c \wedge \neg(\rho\ \sigma\ b)\rangle}$$

$$\frac{c \wedge (\rho\ \sigma\ b) \text{ is satisfiable and } n > 0}{\langle bwhile(n)\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle i; bwhile(n-1)\ b\ i\ ;\ l, \sigma, c \wedge (\rho\ \sigma\ b)\rangle}$$

$$\frac{n = 0}{\langle bwhile(n)\ b\ i\ ;\ l, \sigma, c\rangle \longmapsto \langle ?, \sigma, c\rangle}$$

The CPBPV Bounded Semantics We now define the CPBPV bounded semantics. Let $\mathcal{P}$ be the program $b_{pre}\ l\ b_{post}$ in which b_{pre} denotes the precondition, l is a list of instructions, and b_{post} the post-condition. Let $\overset{*}{\longmapsto}$ be the transitive closure of $\longmapsto$. The final states are specified by the set

$$\mathit{SFN}(b_{pre}, \mathcal{P}) = \{\ \langle f, \sigma, c\rangle | \langle l, \sigma_\perp, \rho\ \sigma_\perp\ b_{pre}\rangle \overset{*}{\longmapsto} \langle f, \sigma, c\rangle\ \wedge\ f \in \{\perp, \top, ?\}\ \}.$$

The program violates an assertion if the set

$$\mathit{SFAE}(b_{pre}, \mathcal{P}, b_{post}) = \{\langle \perp, \sigma, c\rangle | \langle \perp, \sigma, c\rangle \in \mathit{SFN}(b_{pre}, \mathcal{P})\}$$

is not empty. It violates its specification if the set

$$\mathit{SFE}(b_{pre}, \mathcal{P}, b_{post}) = \{\langle \top, \sigma, c\rangle \in \mathit{SFN}(b_{pre}, \mathcal{P})\ |\ c\ \wedge\ (\rho\ \sigma\ \neg b_{post}) \text{ is satisfiable}\}$$

is not empty. It is inconclusive if the set

$$\mathit{SFI}(b_{pre}, \mathcal{P}, b_{post}) = \{\langle ?, \sigma, c\rangle | \langle ?, \sigma, c\rangle \in \mathit{SFN}(b_{pre}, \mathcal{P})\}$$

is not empty. It is totally correct otherwise[8].

3.3 The Verification Algorithm

For completeness, Figure 3 presents the implementation of the CPBPV verifier using concepts directly derived from the bounded semantics. The verifier receives as inputs the initial configuration $\langle l, \sigma_\perp, \rho\ \sigma_\perp\ b_{pre}\rangle$ and the post-condition b_{post}. It returns *success* if the program is totally correct; otherwise, it returns *assertion(s)*, *error(s)*, or *inconclusive(s)* for some configuration s in $\mathit{SFAE}(b_{pre}, \mathcal{P}, b_{post})$, $\mathit{SFE}(b_{pre}, \mathcal{P}, b_{post})$, or $\mathit{SFI}(b_{pre}, \mathcal{P}, b_{post})$ respectively. The algorithm is recursive using configurations $\langle l, \sigma, c\rangle$ reached from the initial configuration in recursive calls.

The basic cases of the recursion considers the inconclusive result (lines 1–2), the assertion error (lines 3–4), and the end of an execution path (lines 5–9). In this last case, either the post-condition is violated (lines 6–7) resulting in an error or the execution path is successful (lines 8–9). Lines 11–16 depicts the main body of the verifier. Line 11 computes the set S of all configurations that can be reached in one step from the current configuration. These configurations are explored recursively and the verifier only succeeds if everyone of these configurations is verified successfully.

[8] In that case, *total* correctness comes from boundness of input data

```
function CPBPVerify($\langle l, \sigma, c \rangle, b_{post}$) =
    if $l = ?$ then
        return inconclusive($\langle l, \sigma, c \rangle$);
    else if $l = \bot$ then
        return assertion($\langle l, \sigma, c \rangle$);
    else if $l = \top$ then
        if $c \ \wedge \ (\rho \ \sigma \ \neg b_{post})$ is satisfiable then
            return error($\langle l, \sigma, c \rangle$);
        else
            return success;
    else
        $S = \{s \ | \ \langle l, \sigma, c \rangle \ \longmapsto \ s\}$;
        forall($s \in S$)
            $r$ = CPBPVerify($s, b_{post}$);
            if $r \neq$ success then
                return $r$;
        return success;
end
```

Fig. 3. The CPBPV Bounded Verification Algorithm.

4 Implementation issues

The CPBPV framework is parametrized by a list of solvers $(S_1, \ldots, S_k)$ which are tried in sequence, starting with the least expensive and least general. When checking satisfiability, the verifier never tries solver $S_{i+1}, \ldots, S_k$ if solver S_i is a decision procedure[9] for the constraint store. If solver S_i is not a decision procedure, it uses an abstraction α of the constraint store c satisfying $c \Rightarrow \alpha$ and can still detect failed execution paths quickly. The last solver in the sequence is a constraint-programming solver (CP solver) over finite domains which iterates pruning and searching to find solutions or prove infeasibility. When the CP solver makes a choice, the earlier solvers in the sequence are called once again to prune the search space or find solutions if they have become decision procedures.

Our prototype implementation uses a sequence (MIP, CP), where MIP is the mixed integer-programming tool ILOG CPLEX[10] and CP is the constraint-programming tool ILOG JSOLVER. Our Java implementation[11] uses the *eclipse*

[9] A solver S is a decision procedure for a constraint store C if and only if it can decide whether a solution of the constraints in C exists. For instance, a solver based on the simplex algorithm is a decision procedure for a set of linear inequalities defined over the rational numbers but not for a set of linear inequalities defined over a finite subset of the integers. However a simplex-based solver may detect that a set of linear inequalities defined over a finite subset of the integers is inconsistent.

[10] See `http://www.ilog.com/products`.

[11] Because ILOG CPLEX and JSOLVER are commercial tools, there is for the moment no public distribution available for CPBPV.

Java Development Tools JDT[12] for parsing the input program, and the API provided on the JML home page[13] for parsing JML specifications. This Java implementation performs some trivial simplifications such as constant propagation but is otherwise not optimized in its use of the solvers and in its renaming process whose speed and memory usage could be improved substantially. Practically, simplifications are performed on the fly and the MIP solver is called at each node of the execution paths. The CP solver is only called at the end of the executable paths when the complete post condition is considered.

More precisely, in the algorithm of Figure 3, we use different heuristics for determining the satisfiability of a constraint store. When testing the correctness of an execution path (line 6), it is necessary to use a complete decision procedure to guarantee correctness. As a result, the CPBPV verifier first calls the MIP solver on the linear subset of the constraint store. If this subset is not consistent, then c is not consistent either. Otherwise, the CP solver is called on the entire constraint store. In contrast, for the transition steps in line 11, it is not strictly necessary to call a complete solver. The CP verifier only calls the MIP solver. If it fails, then no transition takes place. Otherwise, the condition is compatible with the linear subpart of c, but it may be inconsistent with the non-linear part of c. However, the CPBPV verifier does not call the CP solver and continue the exploration of the path although it may be unfeasible. The point is that, since the CP solver may be computationally very demanding, it is usually more efficient to collect more information and to try to reject the path with the MIP solver than to call the CP solver.

The current implementation uses a depth-first strategy for the CP solver, but modern CP languages now offer high-level abstractions to implement other exploration strategies. In practice, when CPBPV is used for model checking as discussed below, it is probably advisable to use a depth-first iterative deepening implementation.

5 Experimental results

In this section, we report some experimental results for a set of classical benchmarks for program verification. We first describe the tools with which we compared CPBPV. We then describe the benchmarks and present the comparative results.

5.1 Existing Tools

CPBPV was compared to five different tools which share the following characteristics:

- They are able to handle the verification of the considered imperative language (see Figure 2), using pre- and post-conditions, even if they are de-

[12] See `http://www.eclipse.org/jdt/`

[13] See `http://www.cs.ucf.edu/~leavens/JML/`

signed to handle more specific programs (e.g., some of the tools can handle CTL/LTL logic formula);
- They perform automatic static analysis;
- A free distribution is available, which we could evaluate on the selected set of benchmarks.

The tools[14] we consider are:

- **CBMC**: a bounded model checker for ANSI-C and C++ programs. It allows the verification of array bounds (buffer overflows), pointer safety, exceptions, and user-specified assertions. See `http://www.cprover.org/cbmc`.
- **EUREKA**: a bounded model checker for ANSI-C which uses an SMT solver instead of an SAT solver. See `http://www.ai-lab.it/eureka`.
- **BLAST**: the Berkeley Lazy Abstraction Software Verification Tool, a software model checker for C programs. See `http://mtc.epfl.ch/software-tools/blast`.
- **ESC/Java**: an Extended Static Checker for Java to find common run-time errors in JML-annotated Java programs by static analysis of the code and its annotations. See `http://kind.ucd.ie/products/opensource/ESCJava2`.
- **Why**: a software verification platform which integrates many existing provers (proof assistants such as Coq, PVS, HOL 4,...) and decision procedures such as Simplify, Yices, ...). See `http://why.lri.fr`.

All of these tools perform automatic static analysis of software to detect programming errors or prove their absence, but they use different techniques[15]:

- CBMC and EUREKA are based on bounded model checking. These tools have some similarities with CPBPV in the sense that they explore the paths reachable within a given number of steps. However, many differences exist, as detailed in Section 6.
- BLAST is based on model-checking and implements an abstraction-refinement process. The main characteristic of BLAST is that it uses *lazzy abstraction*: The refinement step only triggers the relevant parts of the original program [8].
- ESC/JAVA [6, 7] is a static analyzer and requires that the user provides annotations like loop invariants.
- Why is quite different from the previous frameworks in the sense that it aims to perform complete formal verification (it was originally designed to perform fully formal proofs using the Coq theorem prover).

[14] We selected these tools as the most representative but many other tools with similar features exist (e.g, Forge (`http://sdg.csail.mit.edu/forge/` a program analysis framework for checking procedure in a conventional object oriented language, which is based on SAT solver and which is similar to ESC/Java; The KeY System (`http://www.key-project.org/`) a theorem prover for the first-order Dynamic Logic for Java).

[15] See [18] for a recent survey and classification of algorithms and tools to perform automatic static analysis of software.

Despite using (sometimes fundamentally) different techniques, these tools are interesting to evaluate and contrast the capabilities of CPBPV in terms of efficiency, user intervention, and feedback when an error is found.

5.2 Benchmark Programs and Experimental Results

For each benchmark program, we describe the data entries and the verification parameters. More precisely, we give the initial Java program with its JML specification and, when a tool does not accept a similar input because of a lack of expressiveness, we also give the particular input file used within that tool. In the comparative tables, "UNABLE" means that the corresponding tool is unable to validate the program because a lack of expressiveness, "TIME_OUT" because of time or memory limitations, "NOT_FOUND" that it does not detect an error, and "FALSE_ERROR" that it reports an error in a correct program. All experiments were performed on the same computer, an Intel(R) Pentium(R) M processor 1.86GHz with 1.5G of memory, using the version of the verifiers that was downloadable in June 2008 from their web sites (except for EUREKA for which the execution times given in [2, 4] are reported.) More details on these experiments can be found in [21].

Binary search We start with the binary search program presented in Figure 1. ESC/Java, and CBMC require a limit on the number of loop unfoldings, which is set to $\log(n) + 1$, the worst case complexity of the binary search algorithm for an array of length n. ESC/Java was applied on the program described in Figure 1. Since CBMC does not support first-order expressions such as the JML $\backslash forall$ statement, we generated a C program for each instance of the problem (i.e., for each array length). For example, the C program for an array of length 8 is given in Figure 4.

The version of BLAST we used was unable to verify the binary search program because it does not handle nonlinear expressions like the one used to compute the middle index (i.e., mid = (left + right)/ 2). For the Why framework, we used the binary search version given in the Why distribution. This program uses an assert statement to specify a loop invariant and a loop variant as shown in Figure 5.

Note that CPBPV does not require any invariant or a limit on loop unfoldings and thus formally proves correctness with respect to its specification for a given array length. During execution, it selects a path by nondeterministically applying the semantic rules for conditional and loop expressions.

Table 1 reports the experimental results. Execution times for CPBPV are reported as a function of the array length for integers coded with 31 bits.[16] Our implementation is neither optimized for time or space at this stage and times are only given to demonstrate the feasibility of the CPBPV verifier.

The Why framework [24] was unable to verify the correctness without the loop invariant; 60% of the proof obligations remained unknown. CBMC was

[16] The commercial MIP solver fails with 32-bit domains because of scaling issues.

```
int binsearch(int x) {
  int a[8];
  // PRECONDITION
  __CPROVER_assume(a[0]<=a[1]&&a[1]<=a[2]&&a[2]<=a[3]&&a[3]<=a[4]
       &&a[4]<=a[5]&&a[5]<=a[6]&&a[6]<=a[7]);
  signed low=0, high=7;
  int result=-1;
  while(result==-1&&low<=high) {
    signed middle=(high+low)/2;
    if(a[middle]>x)
      high=middle-1;
    else if(a[middle]<x)
      low=middle+1;
    else // a[middle]=x
      result= middle;
  }
  // POSTCONDITION
  assert((result!=-1 && a[result]==x)||(result==-1
     && (a[0]!=x&&a[1]!=x&&a[2]!=x
     &&  a[3]!=x&&a[4]!=x&&a[5]!=x&&a[6]!=x&&a[7]!=x)));
  return result;
}
```

Fig. 4. Input of CBMC used to Verify an Instance of the Binary Search Program for an Array of Length 8.

not able to do the verification for an instance of length 32 (it was interrupted after 6691.87s). ESC/Java was unable to verify the correctness of this program unless complete loop invariants (both on array and index values) are provided. Figure 6 shows a version with loop invariants that has been written by David Cok, a developper of ESC/Java, after we contacted him. This version is verified in 7.36s for 4 unrollings (i.e., an array of length 16).

An Incorrect Binary search Table 2 reports experimental results for an incorrect *binary search* program where a "copy-paste" error has been inserted (see Figure 1, line 11). Again we use CPBPV, ESC/Java using invariant (see Figure 6), CBMC, and Why using an invariant (see Figure 5) . The error trace found with CPBPV has been described in Section 2. The error traces provided by CBMC and ESC/Java only show the decisions taken along the faulty path (i.e., values of left and right indexes). For example, the error trace provided by CBMC is shown in Appendix 1. In contrast to CPBPV, they do not provide any value for the array nor the searched data. Observe that CPBPV provides orders of magnitude improvements in efficiency over CBMC and also outperforms ESC/Java by almost a factor 8 of the largest instance.

The Tritype Program The tritype program is a standard benchmark in test case generation and program verification since it contains numerous non-feasible

```
/*@ requires
  @   n >= 0 && \valid_range(t,0,n-1) &&
  @   \forall int k1, int k2; 0 <= k1 <= k2 <= n-1 => t[k1] <= t[k2]
  @ ensures
  @   (\result >= 0 && t[\result] == v) ||
  @   (\result == -1 && \forall int k; 0 <= k < n => t[k] != v)
  @*/
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1;
  /*@ invariant
    @   0 <= l && u <= n-1 &&
    @   \forall int k; 0 <= k < n => t[k] == v => l <= k <= u
    @ variant u-l
    @*/
  while (l <= u ) {
    int m = (l + u) / 2;
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

Fig. 5. The Binary Search Program with an Invariant from the Why Distribution.

paths: only 10 paths correspond to actual inputs because of complex conditional statements in the program. The program takes three positive integers as inputs (the triangle sides) and returns 2 if the inputs correspond to an isosceles triangle, 3 if they correspond to an equilateral triangle, 1 if they correspond to some other triangle, and 4 otherwise. The tritype program and its specification is shwon in Appendix 2. Note the role of local variable "trityp": It determines how many sides are equal and which are the equal sides.

Table 3 depicts the experimental results for CPBPV, ESC/Java, CBMC, BLAST and Why. Note that, in the data entry for CBMC tool, we replaced each implication $a \Rightarrow b$ by the equivalent disjunction $\neg a \vee b$ because implication is not possible in the assertions. BLAST was unable to validate this example because the version we used does not handle linear arithmetic such as $i+k<=j$. However, it was able to verify an easier version where several cases of the postcondition have been inserted as assertions inside the code at the end of the corresponding paths (see Appendix 3).

Observe the excellent performance of CPBPV, which should be contrasted to our previous approach using constraint programming and Boolean abstraction to abstract the conditions. This earlier approach validated this benchmark in 8.52 seconds when integers were coded with 16 bits [17] and explored 92 spurious paths. This can be easily explained because the tritype program contains many conditionals on input variables that are abstracted as Boolean variables in our previous approach. In particular, there are three successive conditionals whose

CPBPV	array length	8	16	32	64	128	256
	time	1.08s	1.69s	4.04s	17.01s	136.80s	1731.69s
CBMC	array length	8	16	32	64	128	256
	time	1.37s	1.43s	TIME_OUT (>2h)	TIME_OUT	TIME_OUT	TIME_OUT
Why	with invariant	11.18s					
	without invariant	UNABLE					
ESC/Java	FALSE_ERROR						
BLAST	UNABLE						

Table 1. Comparison Table for Binary Search.

	CPBPV	ESC/Java with invariant	CBMC	WHY with invariant
length 8	0.027s	1.644s	1.38s	NOT_FOUND
length 16	0.037s	2.375s	1.69s	NOT_FOUND
length 32	0.064s	2.559s	7.62s	NOT_FOUND
length 64	0.115s	3.067s	27.05s	NOT_FOUND
length 128	0.241s	4.741s	189.20s	NOT_FOUND

Table 2. Experimental Results for an Incorrect Binary Search.

conditions are $i = j$, $i = k$ and $j = k$. When solving the abstract system where $i = j$ (resp. $i = k$ and $j = k$) has been abstracted as the Boolean variable b_0 (resp. b_1 and b_2), the spurious solution $b_0 = 1, b_1 = 1, b_2 = 0$ is found while the integer expressions they represent are inconsistent. On the opposite, if CPBPV takes decision $i = j$ and then decision $i = k$, the consistency test fails for the negation of condition $j = k$ and no spurious path is explored.

An Incorrect Tritype Program Consider now an incorrect version of *Tritype* program in which the test $if((trityp == 2)\&\&(i + k > j))$ in line 22 (see Appendix 2) is replaced by $if((trityp == 1)\&\&(i + k > j))$. Since the local variable *trityp* is equal to *2* when *i==k*, the condition $(i + k) > j$ implies that *(i,j,k)* are the sides of an isosceles triangle (the two other triangular inequalities are trivial because j>0). But, when *trityp=1*, *i==j* holds and this incorrect version may answer that the triangle is isosceles while it may not be a triangle at all. For example, it will return *2* when *(i,j,k)=(1,1,2)*.

Table 4 depicts the experimental results. Execution times correspond to the time required to find the first error. The error trace provided by CPBPV is shown in Appendix 4 and corresponds to input values $(i, j, k) = (1, 1, 2)$ mentioned earlier and thus corresponds to the case where the triangle is isosceles. Other tools (e.g., ESC/JAVA and BLAST) only provide information on the faulty path (see the trace provided by ESC/JAVA in Appendix 4). Once again, observe the excellent behavior of CPBPV compared to the remaining tools. For CBMC, we have contacted D. Kroening who has recommended to use the op-

```
/*@ requires tab != null;
  @ requires (\forall int i,j; (i >= 0 && i < tab.length ) &&
             j >= i && j < tab.length; tab[i] <= tab[j]);
  @ ensures  ((\result == -1) ==> (\forall int i; (i >= 0 &&
                  i < tab.length); tab[i] != x));
  @ ensures  ((\result != -1) ==> (tab[\result] == x));
  @*/
int binarySearch (int[] tab, int x) {
    int result = -1;
    int mid = 0;
    int left = 0;
    int right = tab.length -1;
    //@ maintaining (\forall int i; i>=0 && i < left; tab[i]!=x);
    //@ maintaining (\forall int i; i>right && i < tab.length; tab[i]!=x);
    //@ maintaining result!=-1 ==> tab[result]==x;
    //@ maintaining left>=0;
    //@ maintaining right <= tab.length-1;
    //@ decreases right-left-result;
    while (result == -1 && left <= right) {
        mid = (left + right) / 2;
        if (tab[mid] == x) {
            result = mid;
        } else {
            if (tab[mid] > x) {
                 right = mid - 1;
             } else {
                 left = mid + 1;
             }
        }
      }
    return result;
}
```

Fig. 6. The Binary Search Program with Loop Invariants for ESC/Java.

tion CPROVER_assert. Using this option, CBMC is able to find the error in the faulty version of tritype. However, if we use the same option for the correct tritype program, we must also add the assumption $CPROVER\_assume(i+j >= 0\&\&i+k >= 0\&\&j+k >= 0)$ to ensure that there is no overflow in the sum.

Bubble Sort with initial conditions This benchmark (see Figure 7) is taken from [2] and performs a bubble sort of an array t which contains integers from 0 to $t.length-1$ given in decreasing order. This order corresponds to the worst case complexity of this sort algorithm. However, t contains constant values which makes the problem easier to handle because only one path is possible. Table 5 shows the comparative results for this benchmark. CPBPV was limited on this benchmark because its recursive implementation uses up all the JAVA stack space. This problem should be remedied by removing recursion in CPBPV.

	CPBPV	ESC/Java	CBMC	Why	BLAST	BLAST simplified version
time	0.287s	1.828s	0.82s	8.85s	UNABLE	0.716s

Table 3. Experimental Results on the Tritype Program.

	CPBPV	ESC/Java	CBMC	Why	BLAST	BLAST simplified version
time	0.056s s	1.853s	NOT_FOUND	NOT_FOUND	UNABLE	0.452s

Table 4. Experimental Results for the Incorrect Tritype Program.

Selection Sort We now present a benchmark to highlight both modular verification and the `element` constraint of constraint programming to index arrays with arbitrary expressions. The benchmark is described in Figure 8.

Assume that function `findMin` has been verified for arbitrary integers. When encountering a call to `findMin`, CPBPV first checks if its precondition is entailed by the constraint store, which requires a consistency check of the constraint store with respect to the negation of the precondition. Then CPBPV replaces the call by the post-condition where the formal parameters are replaced by the actual variables. In particular, for the first iteration of the loop and an array length of 40, CPBPV generates the conjunction $0 \leq k^0 < 40 \ \wedge \ t^0[k^0] \leq t^0[0] \ \wedge \ \ldots \ \wedge \ t^0[k^0] \leq t^0[39]$ which features the `element` constraint [35]. Indeed, k^0 is a variable and a constraint like $t^0[k^0] \leq t^0[0]$ indexes the array t^0 of variables using k^0.

The modular verification of the selection sort explores only a single path, is independent of the integer representation, and takes less than $0.01s$ for arrays of size 40. The bottleneck in verifying selection sort is the validation of function `findMin`, which requires the exploration of many paths. However, the complete validation of selection sort takes less than 4 seconds for an array of length 6. Once again, this should be contrasted with the bounded model-checking approach of EUREKA [2] and CBMC: On a version of selection sort where all variables are assigned specific values (contrary to our verification which makes no assumptions on the inputs), EUREKA takes 104 seconds on a faster machine. Reference [2] also reports that CBMC takes 432.6 seconds, that BLAST cannot solve this problem, and that SATABS [14] only verifies the program for an array with 2 elements.

Sum of Squares Our last benchmark is described in Figure 9 and computes the sum of the square of the n first integers stored in an array. The precondition states that n is the size of the array and that t must contain any possible permutation of the n first integers. The postcondition states that the result is $n \times (n + 1) \times (2 \times n + 1)/6$. The benchmark illustrates two functionalities of constraint programming: the ability of specifying combinatorial constraints and

```
/*@ requires (\forall int i; i>=0 && i<t.length;t[i]==t.length-1-i)
 @ ensures  (\forall int i; i>=0 && i<t.length-1;t[i]<=t[i+1]) */
 void bubleSort(int[] t) {
  for (int j = 0; j < t.length; j++)
      for (int i = 0; i< t.length-1; i++)
          if (t[i] > t[i+1]){
             int temp = t[i];
             t[i] = t[i+1];
             t[i+1] = temp;
  }
 }
```

Fig. 7. The Bubble Sort Benchmark From [2].

	CPBPV	ESC/Java	CBMC	EUREKA
length 8	1.45s	3.778 s	1.11s	91s
length 16	2.97s	TIME_OUT	2.01s	TIME_OUT
length 32	TIME_OUT	TIME_OUT	6.10s	TIME_OUT
length 64	TIME_OUT	TIME_OUT	37.65s	TIME_OUT

Table 5. Experimental Results for Bubble Sort.

of solving nonlinear problems. The `alldifferent` constraint [33] in the precondition specifies that all the elements of the array are different, while the program constraints and postcondition involves quadratic and cubic constraints. The maximum instance that we were able to solve with CPBPV was an array of size 10 in 66.179s.

This example highlights one feature of constraint programming, the use of the combinatorial constraint `alldifferent`. Note that we also verified the usual version of this function that takes as input an integer number n and returns the sum of the square of numbers from 0 to n. For that version, the size of the data does not fix the number of loop iterations, which depends on the numeric input value n. Both CPBPV and CBMC were able to verify this program for n less than 128. Their computation time are similar (see Table 6).

Discussion CPLEX, the MIP solver, plays a key role in all these benchmarks since they all contain many linear expressions. For instance, the CP solver is never called in the Tritype benchmark, which only contains linear expressions. For the Binary search benchmark, there are `length` calls to the CP solver, to solve the `element` constraint for the specification case where $result! = -1$ and $t[result] = x$. In that case, both t, $result$ and x are unknown, thus almost 75% of the CPU time is spent in the CP solver. Since there is only one path in the bubbleSort benchmark, the CP solver is only called once. In the sum of squares program, 80% of the CPU time is spent in the CP solver, since here both the specification and the program contains non linear expressions.

```
/*@ ensures  (\forall int i; 0<=i && i<t.length-1;t[i]<=t[i+1]) @*/
   static void selectionSort(int[] t) {
     for (int i=0; i<t.length;i++){
         int k = findMin(t,i);
         int tmp = t[i];
         t[i]= t[k];
         t[k] = tmp;
     }
   }

/*@ requires 0<=l && l<t.length
 @ ensures  (l<=\result) && (\result<t.length)
 @       && (\forall int k; l<=k && k<t.length;t[\result]<=t[k]) @*/
   static int findMin(int[] t,int l) {
     int idx = l;
     for (int j = l+1; j < t.length;j++)
         if (t[idx]>t[j])
            idx = j;
     return idx;
  }
```

Fig. 8. Selection Sort for Modular Verification.

	CPBPV	CBMC
b= 8	0.152s	0.83s
b= 16	0.557s	0.85s
b= 32	1.111s	0.95s
b= 64	1.144s	1.13s
b= 128	1.868s	1.60s

Table 6. Sum of the Squares of the n First Integers.

6 Discussion of Related Work

We briefly review recent work in constraint programming and model checking for software testing, validation, and verification. We outline the main differences between our CPBPV framework and existing approaches.

6.1 Constraint Logic Programming

Constraint logic programming (CLP) was used for test generation of programs (e.g., [25, 29, 34, 27, 3]) and provides a nice implementation tool extending symbolic execution techniques [9]. Gotlieb et al showed how to represent imperative programs as constraint logic programs and used predicate abstraction (from model checking) and conditional constraints within a CLP framework.

Delzanno and Podelski [19] have shown that CLP could provide a conceptual basis for model-checking of infinite-state systems but their implementation

```
\* @ requires (n == t.length-1) && (\forall int i; 0<=i && i<t.length-1;
  @           (\alldifferent t; ) // More compact notation than the
                                  // JML quantified formula
  @ ensures \result == n*(n+1)*(2*n+1)/6 @*/
int sum(int[] t, int n) {
   int s = 0;
   int i = 0;
   while (i!=t.length) {
       s=s+t[i]*t[i]
       i =i+1;    }
   return s;}
```

Fig. 9. Sum of the Squares of the n First Integers in an Array.

mainly uses a BDD-based Boolean solver and a linear solver over reals (integers are abstracted by reals). Flanagan [23] formalized the translation of imperative programs into CLP, argued that it could be used for bounded model checking, but did not provide an implementation.

The test-generation methodology was generalized and applied to bounded program verification in [16, 17]. The implementation was driven by the Boolean solver: a SAT solver was used to solve the Boolean constraint system generated with the information provided by the control flow graph. For each Boolean solution a new constraint system over finite domains was built and solved. The drawback of this approach was the fact that numerous inconsistent constraint systems over finite domains were generated. As shown by the benchmarks in this paper, the strategy of the CPBPV verifier which is based on an incremental detection of inconsistent constraint stores is significantly more efficient.

To sum up, the constraint-programming framework for bounded program verification introduced in this paper is much more scalable and efficient than previous approaches because it is parameterized with a list of solvers (LP solver, MIP solver, Boolean solver, finite domain solver) which are tried in sequence to incrementally prune execution paths.

6.2 Model Checking

It is also useful to contrast the CPBPV verifier with model-checking of software systems. Model checking is an automatic technique for determining if a model of a system satisfies correctness of a specification [18]. Model checking tools have been designed to verify partial specifications, i.e., safety (unreachability of bad states) or liveness properties. Model checkers use a translation to a Boolean representation. A fundamental issue faced by model checkers is the state space explosion of the resulting model. Various techniques have been proposed to address this challenge. The most effective are generalized symbolic execution and abstraction/refinement techniques [30, 18] .

Symbolic model checkers work on implicit representations of sets of states. They can start from initial states as well as from error states. Predicate abstraction is a popular technique to address the state space explosion. The idea consists in abstracting the program to obtain an abstract program on which model checking is performed. The model checker may then generate an abstract counterexample, which must be checked to determine if it corresponds to a concrete execution path. If the counterexample is spurious, the abstract program is refined and the process is iterated. A successful predicate abstraction consists of abstracting the concrete program into a Boolean program (e.g., [10, 12, 13]). In recent work [2], Armando et al proposed to abstract concrete programs into linear programs and used an abstraction of sets of variables and array indices. They showed that their tool compares favourably and, on some of the programs considered in this paper, outperforms model checkers based on predicate abstraction.

Our CPBPV verifier contrasts with SAT-based model checkers using predicate abstraction and refinement techniques: It does not abstract the program and does not generate spurious execution paths. Instead it uses a constraint-solver and nondeterministic exploration to incrementally refine these constraint stores, which define a superset of the potential concrete execution paths. On all the benchmarks of this paper, CPBPV outperforms BLAST, a model checker based on abstraction refinement.

In bounded model checking (BMC), only states reachable within a bounded number of steps are explored. In other words, BMC [11] consists in building a propositional formula whose models correspond to execution paths of bounded length violating some properties and in using SAT solvers to check whether the resulting formula is satisfiable. SAT-based model-checking platforms [11, 28] have been widely popular thanks to significant progress in SAT solvers. The most famous BMC tools are CBMC [20, 13] and F-Soft [28]. They have been designed to handle reachability properties. CBMC was successfully used to compare an ANSI C program with a circuit given as design in Verilog [12]. Armando et al [4] proposed to use SMT[17] solvers instead of SAT solvers for bounded model checking of C programs. They showed that their approach may lead to considerably more compact formulae than those obtained with CBMC.

CPBPV has some similarities with bounded model checking but, in contrast to the above-mentioned BMC tools, CPBPV can easily handle integer data type variables and non-linear relations. To handle floating point variables, we just have to add a dedicated solver (e.g. [9]). On many bounded verification benchmarks, our preliminary experimental results show significant improvements over the state-of-the-art results in [2], SMT-based bounded model checking, and CBMC.

[17] SMT-based bounded model checking is based on the idea of representing and checking quantifier-free formulas in a more general decidable theory (e.g. [26, 22, 31]). SMT solvers integrate dedicated solvers and share some of the motivations of constraint programming. Observe also that this research provides convincing evidence of the benefits of Nieuwenhuis' challenge [31] aiming at extending SMT with CP techniques. See also [1] for a study of the relations between constraint programming and SMT.

7 Perspectives and Future Work

This paper introduced the CPBPV framework for bounded program verification. Its main novelty is to use constraints to represent sets of memory stores and to explore execution paths over these constraint stores nondeterministically and incrementally. As a result, it never explores spurious execution paths contrary to earlier approaches combining constraint programming and predicate abstraction [16, 17] or integrating SMT solvers and the abstraction/refinement approach from model checking [2]. We have demonstrated the CPBPV verifier on a number of standard benchmarks from model checking and program checking as well as on nonlinear programs and functions using complex array indexings, and showed how to perform modular verification. The experimental results demonstrate the potential of the approach: The CPBPV verifier provides significant gain in performance and functionalities compared to other tools.

Our current work aims at improving and generalizing the framework and implementation. In particular, we would like to include tailored, light-weight solvers for a variety of constraint classes, the optimization of the array implementation, and the integration of Java objects and references. There are also many research avenues opened by this research, three of which are reviewed now.

Currently, the CPBPV verifier does not check for variable overflows: the constraint store enforces that variables take values inside their domains and execution paths violating these constraints are thus not considered. It is possible to generalize the CPBPV verifier to check overflows as the verification proceeds. The key idea is to check before each assignment if the constraint store entails that the value produced fits in the selected integer representation and generate an error otherwise. Similar assertions must in fact be checked for each subexpression in the right hand-side in the language evaluation order. Interval techniques on floats [9] may be used to obtain conservative checking of such assertions.

Recent work on loop invariant generation [32] also deserves some attention. Indeed, the use of a priori computed invariant might drastically enhance the scalability of the approach.

An intriguing direction is to use the CPBPV approach for properties checking. Given an assertion to be verified, one may perform a backward execution from the assertion to the function entry point. The negation of the assertion is now the pre-condition and the pre-condition becomes the post-condition. This requires to specify inverse renaming and executions of conditional and iterative statements but these have already been studied in the context of test generation.

Acknowledgements We would like to thank Michel Lecomte for much help and advice with JSOLVER and Jean-François Couchot for advice with the *Why* framework. We are also grateful to David Cok and Joseph Kiriny for discussions about ESC/Java and for providing a version of the binary search.

References

1. Aït-Kaci H., Berstel B., Junker U., Leconte M., Podelski A. Satisfiability Modulo Structures as Constraint Satisfaction : An Introduction. Procs of JFLA 2007.
2. Alessandro Armando, Massimo Benerecetti, Jacopo Mantovani. Abstraction Refinement of Linear Programs with Arrays. Proc. of TACAS 2007: 373-388.
3. Elvira Albert, Miguel Gómez-Zamalloa, Germán Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. Proc. of LOPSTR 2008: 4-23
4. Alessandro Armando, Jacopo Mantovani, Lorenzo Platania. Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. Proc. of Spin 2006: 146-162.
5. Thomas Ball, Sriram K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. Proc. of SPIN 2000: 113-130.
6. Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, Traian Muntean. Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. Proc. of International Workshop, CASSIS 2004, Marseille, France, March 2004, Revised Selected Papers . LNCS (Springer Verlag) 3362:108-128 (2005).
7. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, Erik Poll. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer, 7(3): 212-232 (2005).
8. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar. The Software Model Checker BLAST: Applications to Software. STTT(Journal on Software Tools for Technology Transfer), 9(5-6): 505-525 (2007).
9. Bernard Botella, Arnaud Gotlieb, Claude Michel. Symbolic execution of floating-point computations. Software Testing, Verification and Reliability. 16:2:97–121.2006.
10. Thomas Ball, Andreas Podelski, Sriram K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. Proc. of TACAS 2001:268-283.
11. Edmund M. Clarke, Armin Biere, Richard Raimi, Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design 19(1): 7-34 (2001).
12. Edmund M. Clarke, Daniel Kroening, Flavio Lerda. A Tool for Checking ANSI-C Programs. Proc. of TACAS 2004: 168-176.
13. Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, Karen Yorav. Predicate Abstraction of ANSI-C Programs Using SAT. Formal Methods in System Design 25(2-3): 105-127 (2004).
14. Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, Karen Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. Proc. of TACAS 2005: 570-574.
15. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Trans. Program. Lang. Syst. 13(4): 451-490 (1991).
16. Hélène Collavizza, Michel Rueher. Exploration of the Capabilities of Constraint Programming for Software Verification. Proc. of TACAS 2006: 182-196.
17. Hélène Collavizza, Michel Rueher. Exploring Different Constraint-Based Modelings for Program Verification. Proc. of CP 2007: 49-63
18. Vijay D'Silva, Daniel Kroening, Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. IEEE Trans. on CAD of Integrated Circuits and Systems 27(7): 1165-1178 (2008).

19. Giorgio Delzanno, Andreas Podelski. M odel Checking in CLP. Proc. of TACAS 1999: 223-239
20. Edmund M. Clarke, Daniel Kroening, Karen Yorav. Behavioral consistency of C and verilog programs using bounded model checking. Proc. of DAC 2003: 368-371.
21. Hélène Collavizza, Michel Rueher, Pascal Van Hentenryck. Comparison between CPBPV, ESC/Java, CBMC, Blast, EUREKA and Why for Bounded Program Verification CoRR abs/0808.1508: (2008).
22. Bruno Dutertre, Leonardo Mendonça de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). Proc. of CAV 2006: 81-94.
23. Cormac Flanagan. Automatic software model checking via constraint logic. Sci. Comput. Program. 50(1-3): 253-270 (2004).
24. Jean-Christophe Filliâtre, Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. Proc. of CAV 2007: 173-177.
25. Arnaud Gotlieb, Bernard Botella, Michel Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. Proc. of ISSTA 1998: 53-62.
26. Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, Cesare Tinelli. DPLL(T): Fast Decision Procedures. Proc. of CAV 2004: 175-188.
27. Patrice Godefroid, Michael Y. Levin, David A. Molnar. Automated Whitebox Fuzz Testing. NDSS(Network and Distributed System Security Symposium) 2008.
28. Franjo Ivancic, Zijiang Yang, Malay Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. Theor. Comput. Sci. 404(3): 256-274 (2008).
29. Daniel Jackson, Mandana Vaziri. Finding bugs with a constraint solver. Proc. ISSTA 2000: 14-25.
30. Sarfraz Khurshid, Corina S. Pasareanu, Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. Proc. of TACAS 2003: 553-568.
31. Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, Albert Rubio. Challenges in Satisfiability Modulo Theories. RTA 2007: 2-18.
32. Corina S. Pasareanu, Willem Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. SPIN 2004: 164-181.
33. Jean-Charles Régin. A Filtering Algorithm for Constraints of Difference in CSPs. AAAI 1994: 362-367.
34. Nguyen Tran Sy, Yves Deville. Automatic Test Data Generation for Programs with Integer and Float Variables. ASE 2001: 13-21.
35. Pascal Van Hentenryck. Constraint Satisfaction in Logic Programming. MIT Press, 1989.
36. Pascal Van Hentenryck, Laurent Michel, Yves Deville. Numerica: A Modeling Language for Global Optimization. MIT Press, 1997.

Appendices

Appendix 1: Error trace provided by CBMC for an incorrect binary search

```
Counterexample:
State 15 file bsearchAssertKO.c line 10 function binsearch thread 0
----------------------------------------------------
  bsearchAssertKO::binsearch::1::low=0 (00000000000000000000000000000000)
State 16 file bsearchAssertKO.c line 10 function binsearch thread 0
----------------------------------------------------
  bsearchAssertKO::binsearch::1::high=7 (00000000000000000000000000000111)
```

```
State 17 file bsearchAssertKO.c line 11 function binsearch thread 0
----------------------------------------------------
  bsearchAssertKO::binsearch::1::result=-1 (11111111111111111111111111111111)
State 18 file bsearchAssertKO.c line 13 function binsearch thread 0
----------------------------------------------------
  bsearchAssertKO::binsearch::1::1::middle=3 (00000000000000000000000000000011)
State 21 file bsearchAssertKO.c line 17 function binsearch thread 0
----------------------------------------------------
  bsearchAssertKO::binsearch::1::high=2 (00000000000000000000000000000010)
State 25 file bsearchAssertKO.c line 13 function binsearch thread 0
----------------------------------------------------
  bsearchAssertKO::binsearch::1::1::middle=1 (00000000000000000000000000000001)
State 29 file bsearchAssertKO.c line 15 function binsearch thread 0
----------------------------------------------------
  bsearchAssertKO::binsearch::1::high=0 (00000000000000000000000000000000)
State 33 file bsearchAssertKO.c line 13 function binsearch thread 0
----------------------------------------------------
  bsearchAssertKO::binsearch::1::1::middle=0 (00000000000000000000000000000000)
State 37 file bsearchAssertKO.c line 15 function binsearch thread 0
----------------------------------------------------
  bsearchAssertKO::binsearch::1::high=-1 (11111111111111111111111111111111)
Violated property:
  file bsearchAssertKO.c line 21 function binsearch
  assertion
  result != -1 && a[result] == x || result == -1 && a[0] != x && a[1] != x
      && a[2] != x && a[3] != x && a[4] != x && a[5] != x && a[6] != x && a[7] != x
VERIFICATION FAILED
```

Appendix 2: the Tritype program

```
/*@ requires (i>=0)&&(j>=0)&&(k>=0);
 @ ensures
 @   ((i+j<=k)||(j+k<=i)||(i+k<=j)) ==> \result == 4 &&
 @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&((i==j)&&(j==k)) ==> \result == 3 &&
 @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
 @        &&((i==j)||(j==k)||(i==k)) ==> \result == 2 &&
 @   !((i+j<=k)||(j+k<=i)||(i+k<=j))&&!((i==j)&&(j==k))
 @        &&!((i==j)||(j==k)||(i==k)) ==> \result == 1;
@*/
   public static int tritype(int i, int j, int k){
    int trityp ;
    // not a triangle
    if ((i==0)||(j==0)||(k==0)) trityp = 4 ;
    else {
      trityp = 0 ;
      if (i==j) trityp = trityp + 1 ;
      if (i==k) trityp = trityp + 2 ;
      if (j==k) trityp = trityp + 3 ;
      if (trityp==0){
         // triangular inequality not verified
         if ((i+j <= k)||(j+k <= i)||(i+k <= j)) trityp = 4 ;
         else trityp = 1 ; // any triangle
      }
      else {
        if (trityp > 3) trityp = 3 ; // equilateral
        else
          //i=j and triangular inequality verified
          if ((trityp==1)&&(i+j>k)) trityp = 2 ;
          else
            //i=k and triangular inequality verified
            if ((trityp==2)&&(i+k>j)) trityp = 2 ;
                            //ERROR if ((trityp==1)&&(i+k>j))
            else
              //j=k and triangular inequality verified
              if ((trityp==3)&&(j+k>i)) trityp = 2 ;
              else trityp = 4 ; // not a triangle
            }
        }
      return trityp;
    }
```

Appendix 3: Version of tritype program verified with BLAST

```
#include <assert.h>
int main(int i, int j, int k)  {
     int trityp ;
     if ((i <= 0) || (j <= 0) || (k <= 0)){
        trityp = 4 ;
        assert((i <= 0) || (j <= 0) || (k <= 0));
     }
     else {
         trityp = 0 ;
         if (i == j)  trityp = trityp + 1 ;
        if (i == k)  trityp = trityp + 2 ;
         if (j == k ) trityp = trityp + 3 ;
         if (trityp == 0)  {
            if ((i+j <= k) || (j+k <= i) || (i+k <= j)) {
                trityp = 4 ;
                assert((i+j<=k)||(j+k<=i)||(i+k<=j));
            }
            else{
                trityp = 1 ;
                assert((i!=j) && (j!=k) && (i!=k)
                        && !((i+j<=k)||(j+k<=i)||(i+k<=j))  );
            }
```

```
       }
       else {
            if (trityp > 3) {
              trityp = 3 ;
              assert((i==j && j==k && i==k));
            }
            else
              if ((trityp == 1) && (i+j > k) ){
                 trityp = 2 ;
                 assert(i==j );
              }
              else
                if ((trityp == 2) && (i+k > j) ){ //ERROR trityp==1
                   trityp = 2 ;
                   assert(i==k );
                }
                else
                  if ((trityp == 3) && (j+k > i)) {
                     trityp  = 2 ;
                     assert(j==k);
                  }
                  else {
                     trityp = 4 ;
                     assert((i+j<=k)||(j+k<=i)||(i+k<=j));
                  }
         }
    }
  return trityp ;
}
```

Appendix 4: Error trace provided by ESC/JAVA and CPBPV for faulty tritype program

Error trace provided by CPBPV

*trityp*_3 is the last renaming of local variable “trityp”

```
i_0[-2147483647:2147483646] : 1
j_0[-2147483647:2147483646] : 1
k_0[-2147483647:2147483646] : 2
trityp_0[-2147483647:2147483646] : 0
trityp_1[-2147483647:2147483646] : 0
trityp_2[-2147483647:2147483646] : 1
trityp_3[-2147483647:2147483646] : 2
```

Error trace provided by ESC/JAVA

```
TritypeKO.java:67: Warning: Postcondition possibly not established (Post)
        }
        ^
Associated declaration is "TritypeKO.java", line 12, col 5:
          @ ensures ...
            ^
Execution trace information:
    Executed else branch in "TritypeKO.java", line 23, col 7.
    Executed then branch in "TritypeKO.java", line 25, col 15.
    Executed else branch in "TritypeKO.java", line 28, col 3.
    Executed else branch in "TritypeKO.java", line 31, col 3.
    Executed else branch in "TritypeKO.java", line 42, col 8.
    Executed else branch in "TritypeKO.java", line 46, col 9.
    Executed else branch in "TritypeKO.java", line 50, col 10.
    Executed then branch in "TritypeKO.java", line 51, col 39.
    Executed return in "TritypeKO.java", line 66, col 2.
```

```
Counterexample context:
    (0 < k:18.32)
    ((2 * j:18.25) <= k:18.32)
    (k:18.32 <= intLast)
    (longFirst < intFirst)
    (1000001 <= intLast)
    (null <= max(LS))
    (eClosedTime(elems) < alloc)
    (vAllocTime(this) < alloc)
    ((intFirst + 1000001) <= 0)
    (intLast < longLast)
    (0 <= j:18.25)
    (k:18.32 == 0) == tmp0!cor:20.6
    null.LS == @true
    (null <= max(LS))
    typeof(j:18.25) <: T_int
    ((j:18.25 + k:18.32) > j:18.25) == @true
    (0 + 1) == 1
    (j:18.25 == 0) == tmp1!cor:20.6
    typeof(k:18.32) <: T_int
    typeof(this) <: T_TritypeKO
    ((j:18.25 + j:18.25) > k:18.32) == tmp4!cand:47.9
    typeof(this) <: T_TritypeKO
    trityp:19.6<7> == 2
    T_bigint == T_long
    tmp0!cor:20.23 == tmp0!cor:20.6
    trityp:19.6<2> == 1
    trityp:19.6<5> == 2
    elems@pre == elems
    j:18.25 == i:18.18
    trityp:19.6<8> == 2
    tmp5!cand:51.25 == @true
    trityp:19.6 == 2
    trityp:26.4 == 1
    trityp:19.6<3> == 1
    state@pre == state
    trityp:19.6<6> == 2
    tmp1!cor:20.13 == tmp1!cor:20.6
    trityp:19.6<1> == 1
    tmp5!cand:51.13 == @true
    alloc@pre == alloc
    tmp4!cand:47.21 == tmp4!cand:47.9
    !typeof(this) <: T_void
    !T_java.lang.Object <: T_java.io.Serializable
    typeof(this) != T_void
    bool$false != @true
    tmp4!cand:47.9 != @true
    ecThrow != ecReturn
    1 != 0
    k:18.32 != j:18.25
    k:18.32 != 0
    this != null
    trityp:19.6<7> != 4
    tmp0!cor:20.23 != @true
    j:18.25 != 0
    tmp1!cor:20.6 != @true
```