CPBPV: A Constraint-Programming Framework For Bounded Program Verification

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

[1] Université de Nice–Sophia Antipolis, France ({helen,rueher}@polytech.unice.fr)
[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 the specification and the program and explores execution paths nondeterministically. The input program is partially correct 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 parametrized 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 variable domains. Moreover, CPBPV was able to detect subtle errors in some programs while other frameworks based on 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), i.e., when the program inputs (e.g., the array lengths and the variable values) are bounded. 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. The key idea of CPBPV is to use constraint stores to represent the specification and the program, and to non-deterministically explore execution paths over these constraint stores. This non-deterministic constraint-based symbolic execution incrementally refines the constraint store, which initially consists of the precondition. Non-determinism occurs when executing conditional or iterative instructions and the non-deterministic execution refines the constraint store by adding constraints coming from conditions and from assignments. The input program is partially correct if each constraint store produced by the symbolic execution implies the post-condition. It is important to emphasize that CPBPV considers programs with complete specifications and 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 model-checking tools designed to detect violations of some specific property, e.g., safety or liveness properties.

The CPBPV framework has a number of fundamental benefits. First, contrary to earlier work using constraint programming or SMT [2, 12, 13], CPBPV does not use predicate abstraction or 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 combination 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 less general.

The CPBPV framework was evaluated experimentally on a series of benchmarks from program verification. Experimental results of our (slow) 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 was able to found subtle errors in some programs that some other verification frameworks based on model-checking could not detect.

The rest of the paper is organized as follows. Section 2 illustrates how CPBPV handles constraints store on a motivating example. Section 3 formalizes the CPBPV framework for a small 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 model-checking based 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.

2 The Constraint-Programming Framework at Work

This section illustrates the CPBPV verifier on a motivating example, the binary search program. CPBPV uses Java programs and JML specifications for the pre- and post-conditions, appropriately enhanced to support the expressivity of constraint programming. Figure 1 depicts a binary search program to determine if 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 on a number of bits.

The initial constraint store of the CPBPV verifier, assuming an input array of length 8, is the precondition[3]

$$c_{pre} \equiv \forall 0 \leq i < 7 : t^0[i] \leq t^0[i+1]$$

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

```
/*@ 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 else u = m - 1;
12   return -1; }
```

Fig. 1. The Binary Search Program

where t^0 is an array of constraint variables capturing the input. The constraint variables are annotated with a version number as CPBPV performs a SSA-like renaming [11] 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 $l^0 \leq u^0$, it enters the loop body, adds the constraint $m^0 = (l^0 + u^0)/2$, which simplifies to $m^0 = 3$, and considers the conditional statement on line 6. 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, so that 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 an 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 computation 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 $u^1 = m^0 - 1$. 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 stores entails $l^0 \leq u^1$, the only extension to the current path consists of 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, $\langle T_4, F_6, T_8, T_5, T_6 \rangle$ 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 instruction of instruction 8, the true alternative of the second execution of instruction 5, 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 on 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). 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. The verifier returns a counter-example for debugging the program.

Note that *CBMC* and *ESC/Java*, two state-of-the-art model checkers fail to verify this example as discussed in Section 5.

3 Formalization of the Framework

This section formalizes the CPBPV verifier on a small abstract language using a big-step SOS semantics. The semantics primarily specifies the execution paths over constraint stores explored by the verifier. It features `assert` and `enforce` constructs which are necessary for modular composition.

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.

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

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

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*

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

Fig. 2. The Syntax of Programs and Constraints

Renamings CPBPV creates variables and arrays of variables "on-the-fly" when they are needed. This process resembles an SSA 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 a 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 now maps to i, i.e.,

$$\sigma[x/i](y) = \mathit{if} x = y \ \mathit{then}\ i \ \mathit{else}\ \sigma(y).$$

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.

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 $\langle \perp, \sigma, c\rangle$ 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.

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.

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

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 $\neg c_b$.

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

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

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.

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

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 which may contain variables as well, giving rise to the well-known *element* constraint in constraint programming [26].

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

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.

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

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

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

Block Statements Block statements simply remove the braces.

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

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

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

Termination Termination also occurs when no instruction remains.

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

The CPBPV Semantics Let $\mathcal{P}$ be 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

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

The program violates an assertion if the set

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

is not empty. It violates its specification if the set

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

is not empty. It is partially correct otherwise.

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 less general. When checking satisfiability, the verifier never tries solver $S_{i+1}, \ldots, S_k$ if solver S_i is a decision procedure 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[4] and CP is the constraint-programming tool Ilog JSOLVER. Our Java implementation also 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

[4] See http://www.ilog.com/products.

improved substantially. The implementation use a depth-first strategy currently, 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 experimental results for a set of traditional benchmarks for program verification. We compare CPBVP with the following frameworks:

- ESC/Java is 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/.
- CBMC is a Bounded Model Checker for ANSI-C and C++ programs. It allows for the verification of array bounds (buffer overflows), pointer safety, exceptions, and user-specified assertions. See http://www.cprover.org/cbmc/.
- BLAST, the Berkeley Lazy Abstraction Software Verification Tool, is a software model checker for C programs. See http://mtc.epfl.ch/software-tools/blast/.
- EUREKA is a C bounded model checker which uses an SMT solver instead of an SAT solver. See (http://www.ai-lab.it/eureka/.
- Why is 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 experiments were performed on the same machine, an Intel(R) Pentium(R) M processor 1.86GHz with 1.5G of memory, using the version of the verifiers that can be downloaded from their web sites (except for EUREKA for which the execution times given in [2, 3] are reported.) For each benchmark program, we describe the data entries and the verification parameters. In the tables, "UNABLE" means that the corresponding framework is unable to validate the program either because a lack of expressiveness or 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. Complete details of the experiments, including input files and error traces, can be found in [14].

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

CPBPV	array length	8	16	32	64	128	256
	time	1.081s	1.69s	4.043s	17.009s	136.80s	1731.696s
CBMC	array length	8	16	32	64	128	256
	time	1.37s	1.43s	UNABLE	UNABLE	UNABLE	UNABLE
Why	with invariant	11.18s					
	without invariant	UNABLE					
ESC/Java	FALSE_ERROR						
BLAST	UNABLE						

Table 1. Comparison table for binary search

```
(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)
```

For the Why framework, we used the binary search version given in their distribution. This program uses an assert statement to give a loop invariant.

Note that CPBPV does not require any additional information: no invariant and no limits on loop unfoldings. 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 on 31 bits.[5] 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 [17] was unable to verify the correctness without the loop invariant; 60% of the proof obligations remained unknown.

The CBMC framework was 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 are provided. [6]

An Incorrect Binary search Table 2 reports experimental results for an incorrect *binary search* program. (see Figure 1, line 11) for CPBPV, ESC/Java, CBMC, and Why using an invariant. 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 (see Figure 7 in the Appendix). 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 on 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 paths: only 10 paths correspond to actual inputs because of complex conditional

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

[6] a version with loop invariants that allows to show the correctness of this program has been written by David Cok, a developper of ESC/Java, after we contacted him.

	CPBPV	ESC/Java	CBMC	WHY with invariant	BLAST
length 8	0.027s	1.21 s	1.38s	NOT_FOUND	UNABLE
length 16	0.037s	1.347 s	1.69s	NOT_FOUND	UNABLE
length 32	0.064s	1.792 s	7.62s	NOT_FOUND	UNABLE
length 64	0.115s	1.886 s	27.05s	NOT_FOUND	UNABLE
length 128	0.241s	1.964 s	189.20s	NOT_FOUND	UNABLE

Table 2. Experimental Results for an Incorrect Binary Search

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

Table 3. Experimental Results on the Tritype Program

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. Figure 6 in the Appendix gives the `tritype` program in Java with its specification in JML. Table 3 depicts the experimental results for CPBPV, ESC/Java, CBMC, BLAST and Why. BLAST was unable to validate this example because the current version does not handle linear arithmetic. Observe the excellent performance of CPBPV and note that our previous approach using constraint programming and Boolean abstraction to abstract the conditions, validated this benchmark in 8.52 seconds when integers were coded on 16 bits [13]. It also explored 92 spurious paths.

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 Figure 6 in the Appendix) 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 found with CPBPV corresponds to input values $(i, j, k) = (1, 1, 2)$ mentioned earlier. Once again, observe the excellent behavior of CPBPV compared to the remaining tools. [7]

[7] For CBMC, we have contacted D. Kroening who has recommended to use the option CPROVER_assert. If we do so, CBMC is able to find the error, but we must add some assumptions to mean that there is no overflow into the sums, in order to prove the correct version of tritype with this same option.

	CPBPV	ESC/Java	CBMC	WHY
time	0.056s s	1.853s	NOT_FOUND	NOT_FOUND

Table 4. Experimental Results for the Incorrect Tritype Program

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

Table 5. Experimental Results for Bubble Sort

Bubble Sort with initial condition This benchmark, depicted in Figure 3 (Annex 1), is taken from [2] and performs a bubble sort of an array t which contains integers from 0 to $t.length$ given in decreasing order. 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.

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 depicted in Figure 4 in the Appendix. 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 `element` constraint [26]. 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 model-checking approach of Eureka [2]. 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 [10] only verifies the program for an array with 2 elements.

Sum of Squares Our last benchmark is depicted in Figure 5 in the Appendix 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 of solving nonlinear problems. The `alldifferent` constraint[24] 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.

6 Discussion and 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.

Constraint Logic Programming Constraint logic programming (CLP) was used for test generation of programs (e.g., [18, 21, 25, 20]) and provides a nice implementation tool extending symbolic execution techniques [5]. 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. Flanagan [16] 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 [12, 13]. The implementation used dedicated predicate abstractions to reduce the exploration of spurious execution paths. However, as shown in the paper, the CPBPV verifier is significantly more efficient and often avoids the generation of spurious execution paths completely.

Model Checking It is also useful to contrast the CPBPV verifier with model-checking of software systems. SAT-based bounded model checking for software (e.g., [7]) 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 [4, 7] have been widely popular thanks to significant progress in SAT solvers. 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, including generalized symbolic execution (e.g., [22]), SMT-based model checking, and abstraction/refinement techniques. SMT-based model checking is the idea of representing and checking quantifier-free formulas in a more general decidable theory (e.g. [19, 15, 23]). These SMT solvers integrate dedicated solvers and share some of the motivations of constraint programming. Predicate abstraction is another 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., [6, 8, 9]). In recent work [3, 2], Armando & 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, SMT-based model checkers and predicate abstraction based approaches: It does not abstract the program and does not generate spurious execution paths. Instead it uses a constraint-solver and nondeterministic exploration to incrementally construct abstractions of execution paths. The abstraction uses constraint stores to represent sets of concrete stores. On many bounded verification benchmarks, our preliminary experimental results show significant improvements over the state-of-the-art results in [2]. Model checking is well adapted to check low-level C program and hardware applications with numerous Boolean constraints and bit-wise operations: It was successfully used to compare an ANSI C program with a circuit given as design in Verilog [8]. However, it is important to observe that in model checking, one is typically interested in checking some specific properties such as buffer overflows, pointer safety, or user-specified assertions. These properties are typically much less detailed than our post-conditions and abstracting the program may speed up the process significantly. In our CPBPV verifier, it is critical to explore all execution paths and the main issue is how to effectively abstract memory stores by constraints and how to check satisfiability incrementally. It is an intriguing issue to determine whether an hybridization of the two approaches would be beneficial for model checking, an issue briefly discussed in the next section. Observe also that this research provides convincing evidence of the benefits of Nieuwenhuis' challenge [23] aiming at extending SMT[8] with CP techniques.

7 Perspectives and Future Work

This paper introduced the CPBPV framework for bounded program verification. Its novelty is to use constraints to represent sets of memory stores and to explore

[8] See also [1] for a study of the relations between constraint programming and Satisfiability Modulo Theories (SMT)

execution paths over these constraint stores nondeterministically and incrementally. The CPBPV verifier exploits the fact that, when variables and arrays are bounded, the constraint store can always be checked for feasibility. As a result, it never explores spurious execution path contrary to earlier approaches combining constraint programming and predicate abstraction [12, 13] or integrating SMT solvers and the abstraction/refinement approach from model checking [2]. We 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, two 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 [5] may be used to obtain conservative checking of such assertions.

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 Many thanks to Jean-Franois Couchot for many helps on the use of the *Why framework.*

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. *Armando A., Benerecetti M., and Montovani J. Abstraction Refinement of Linear Programs with Arrays. Proceedings of TACAS 2007, LNCS 4424: 373–388.*
3. *Armando A., Mantovani J., and Platania L. Bounded Model Checking of C Programs using a SMT solver instead of a SAT solver. Proc. SPIN'06. LNCS 3925, Pages 146-162.*

4. *Thomas Ball, Sriram K. Rajamani: Bebop: A Symbolic Model. Checker for Boolean Programs. SPIN 2000: 113-130.*
5. *Botella B., Gotlieb A., Michel C. Symbolic execution of floating-point computations. Software Testing, Verification and Reliability. 16:2:97–121.2006.*
6. *Thomas Ball, Andreas Podelski, Sriam K. Rajamani Boolean and Cartesian Abstraction for Model Checking C Programs. Proc. of TACAS 2001.*
7. *E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking using Satisfiability Solving. Formal Methods in System Design, 19(1):7–34, 2001.*
8. *Clarke E., Kroening D., Lerda F. : A Tool for Checking ANSI-C programs. Tacas 2004, LNCS 2988, pp 168-176, 2004*
9. *Clarke E., Kroening D., Sharygina N., Yorav K. : Predicate abstraction of ANSI-C Programs using SAT. Formal Methods in System Design, Vol* ***25****, pp 105-127, Kluwer Academic Press, 2004*
10. *Clarke E., Kroening D., Sharygina N., Yorav K. : SATABS: SAT-Based Predicate Abstraction for ANSI-C. TACAS'05, 570–574, 2005.*
11. *Cytron R., Ferrante J., Rosen B., Wegman M., and Zadeck K. : Efficently Computing Static Single Assignment Form and the Control Dependence Graph.* Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.*
12. *Collavizza H. and Rueher M. : Software Verification using Constraint Programming Techniques. Procs of TACAS 2006, LNCS 3920: 182-196.*
13. *Collavizza H. and Rueher M. : Exploring different constraint-based modelings for program verification Procs of CP 2007, LNCS 3920: 182-196*
14. *Collavizza H. Rueher M., Van Hentenryck P. : Comparison between CPBPV with ESC/Java, CBMC, Blast, EUREKA and Why. http://www.i3s.unice.fr/˜rueher/verificationBench.pdf*
15. *Bruno Dutertre and Leonardo Mendonca de Moura. A fast linear-arithmetic solver for DPLL(T). CAV 2006, pages 81–94. LNCS 4144.*
16. *Cormac Flanagan, "Automatic software model checking via constraint logic" (2004). Science of Computer Programming. 50 (1-3), pp. 253-270.*
17. *Fillitre J.C., Claude March.The Why/Krakatoa/Caduceus Platform for Deductive Program Verification Proc. CAV'2007, LNCS 4590. pp 173-177.*
18. *Gotlieb A., Botella B. and Rueher M : Automatic Test Data Generation using Constraint Solving Techniques. Proc. ISSTA 98, ACM SIGSOFT (2), 1998.*
19. *Ganzinger H., Hagen G., Nieuwenhuis R.,Oliveras A., and Tinelli C.: DPLL(T): Fast Decision Procedures. Proc. of CAV 2004, 175-188, 2004.*
20. *P. Godefroid, M. Y. Levin, D. Molnar: Automated Whitebox Fuzz Testing, NDSS 2008, Network and Distributed System Security Symposium.*
21. *Daniel Jackson and Mandana Vaziri, Finding Bugs with a Constraint Solver, ACM SIGSOFT Symposium on Software Testing and Analysis, 14–15, 2000.*
22. *Khurshid, S., Pasareanu, C.S., and Vissser, W. "Generalized Symbolic Execution for Model Checking and Testing", in TACAS 2003, Warsaw, Poland.*
23. *R. Nieuwenhuis, A. Oliveras, E. Rodrguez-Carbonell and A. Rubio: Challenges in Satisfiability Modulo Theories. Invited Talk. RTA 2007, LNCS 4533, pp 2-18.*
24. *J-C. Régin. A filtering algorithm for constraints of difference in CSPs. AAAI-94, Seattle, WA, USA, pp 362–367, 1994.*
25. *Sy N.T. and Deville Y.: Automatic Test Data Generation for Programs with Integer and Float Variables. Proc of. 16th IEEE ASE01, 2001.*
26. *VanHentenryck P. (1989) Constraint Satisfaction in Logic Programming, MIT Press.*
27. *Numerica: A Modeling Language for Global Optimization Pascal Van Hentenryck, Laurent Michel, Yves Deville. MIT Press, 1997.*

Annex 1: Some of the Benchmark Programs

```
/*@ 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. 3. The Bubble Sort Benchmark From [2]

```
/*@ 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. 4. Selection Sort for Modular Verification

```
\* @ requires (n == t.length-1) && (\forall int i; 0<=i && i<t.length-1;
  @           (\alldifferent t; ) // More compact notation than the JML  quantified formulae
  @ 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. 5. Sum of the square of the n first integers

```
/*@ 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;
 }
```

Fig. 6. The Tritype Program

```
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
```

Fig. 7. Error trace given by CBMC for binary search with error