



LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

SEMANTICALLY-DRIVEN BOUNDED MODEL CHECKING USING THEOREM PROVING, SMT AND CONSTRAINT SOLVING

Hélène Collavizza, Mike Gordon

Equipe CEP

Rapport de recherche
ISRN I3S/RR-2009-13-FR

Septembre 2009

RÉSUMÉ :

Nous proposons une méthode de bounded model-checking pour la vérification des logiciels qui combine un assistant de preuves en logique d'ordre supérieur (HOL) et des procédures de décision. La méthode assure un bon compromis entre l'efficacité et l'assurance de correction. A partir d'un état symbolique initial et d'une pré-condition, un programme est exécuté symboliquement le long de tous les chemins exécutables jusqu'à une certaine profondeur donnée. Ensuite une post-condition est prouvée ou réfutée à l'état final. Chaque étape d'exécution symbolique est effectuée à partir de théorèmes qui définissent en HOL la sémantique formelle du langage de programmation. Un solveur SMT externe (pour les formules du premier ordre) et de résolution de contraintes (pour les expressions non linéaires) coupent les chemins d'exécution quand les conditions ne sont pas consistantes avec l'état courant. À la fin de chaque chemin, la post-condition est vérifiée en essayant d'abord une preuve déductive puis par appel aux solveurs externes. L'utilisation de théorèmes dérivés de la sémantique formelle du langage assure que chaque instruction est correctement exécutée. Les solveurs externes accroissent l'efficacité. Sur un ensemble d'exemples simples, la plupart des preuves ont été effectuées de façon automatique par le démonstrateur de théorèmes, les solveurs externes sont principalement utilisés pour calculer les contre-exemples et résoudre les expressions non linéaires.

MOTS CLÉS :

vérification des programmes, contraintes, SMT solveur, HOL

ABSTRACT:

We propose a combination of theorem proving and bounded model checking as a method for software verification that supports flexible assurance/efficiency tradeoffs. Starting from an initial symbolic state and a precondition, a program is symbolically executed along all feasible paths up to some given depth and then a postcondition is proved or refuted on the final states. Each symbolic execution step is performed using theorems proved from a formal semantics of the programming language represented in higher order logic. An external SMT solver (for first order formulae) and constraint solver (for non-linear expressions) prunes execution paths by testing whether conditions are feasible in the current state. At the end of each path, the postcondition is verified by trying deductive theorem proving and external solvers in sequence. Using theorems derived from the language semantics assures that each command is symbolically executed correctly. The external solvers boost efficiency, but are not trusted. Theorems proved with external solvers are tagged to indicate how the theorem was proved and the tag is propagated whenever the theorem is used so that the trustworthiness of each link in a verification is apparent. On simple examples, most of the proofs are automatically done by the theorem prover, the external solvers are mainly used to compute counter-examples and check non-linear expressions.

KEY WORDS :

bounded model checking, constraints SMT solver, HOL

Semantically-driven Bounded Model Checking using Theorem Proving, SMT and Constraint Solving

Hélène Collavizza¹ and Mike Gordon²

¹ Université de Nice–Sophia-Antipolis – I3S/CNRS, 930, route des Colles
B.P. 145 06903 Sophia-Antipolis, France. helen@polytech.unice.fr

² University of Cambridge, Computer Laboratory, William Gates Building, 15 JJ
Thomson Avenue, Cambridge CB3 0FD, UK. Mike.Gordon@cl.cam.ac.uk

Abstract. We propose a combination of theorem proving and bounded model checking as a method for software verification that supports flexible assurance/efficiency tradeoffs. Starting from an initial symbolic state and a precondition, a program is symbolically executed along all feasible paths up to some given depth and then a postcondition is proved or refuted on the final states. Each symbolic execution step is performed using theorems proved from a formal semantics of the programming language represented in higher order logic. An external SMT solver (for first order formulae) and constraint solver (for non-linear expressions) prunes execution paths by testing whether conditions are feasible in the current state. At the end of each path, the postcondition is verified by trying deductive theorem proving and external solvers in sequence.

Using theorems derived from the language semantics assures that each command is symbolically executed correctly. The external solvers boost efficiency, but are not trusted. Theorems proved with external solvers are tagged to indicate how the theorem was proved and the tag is propagated whenever the theorem is used so that the trustworthiness of each link in a verification is apparent. On simple examples, most of the proofs are automatically done by the theorem prover, the external solvers are mainly used to compute counter-examples and check non-linear expressions.

1 Introduction

Methods for the formal verification of software can be conducted at a range of degrees of formality. At one end of the spectrum everything is coded without any mechanical link to formal specifications. The formal semantics of the programming language is just documentation and the only link between it and the verifier code is in the mind of the programmer. At the other extreme everything is mechanically deduced from a formalisation of the language semantics. The advantage of deriving everything from a semantics is high assurance of verification soundness. The disadvantage is performance: computation by deduction in a theorem prover cannot compete with efficient algorithms coded in (usually unsafe) languages.

The work described here aims to achieve both good assurance and efficiency by taking a middle approach that combines external oracles with deductively assured execution models.

We exploit theorems derived from a formal programming language semantics using a theorem prover (HOL4) to specify state transitions for bounded model checking (see [8] for a recent and complete survey of BMC).

Efficiency is achieved by using fast automatic external solvers as oracles for pruning infeasible execution paths and verifying that final states entail desired postconditions. The theorem prover is used to generate the transition function and, in addition, it provides simplification tools and decision procedures which are more powerful (though much slower) than those provided by the external solvers. Our methodology tries fast methods first – an overview follows.

- BMC consists of symbolic execution of all feasible paths up to a given depth, followed by checking that the state at the end of the path entails a desired postcondition (different paths corresponding to different choices at conditional branches).
- For each symbolic execution step, the next state is computed using a state-transition function derived by theorem proving from the formal semantics applied to the program. This provides assurance that each command is symbolically executed according to its formal semantics.
- external solvers (an SMT solver for linear expressions and a constraint solver for non-linear expressions) are called when conditional branches are reached to check if the formula representing the state determines the condition. If it does then only one path need be taken, the other being infeasible. If the conditional branch can't be resolved then paths from both branches must be verified.
- At the end of each path, deductive theorem proving and external solvers are tried in sequence. The solvers provide a decision procedure for a finite subset of integers. As each path is verified separately, the verification formulae do not contain any conditional branches so are relatively small and it is thus feasible to apply deductive theorem proving, which is not tailored to handle enormous formulae with a complex Boolean structure (this contrasts with the capabilities of SAT or SMT solvers which are generally used in BMC tools [8, 14, 5, 2]).
- If the verification fails, external solvers can efficiently compute counter-examples.

The rest of the paper is structured as follows: first a simple example is used to illustrate key ideas. Then we present the formalisation of the programming language semantics and show how derived theorems are used to perform part of computation. Next we explain how we combine external solvers with the theorem prover (HOL4). We then describe the symbolic execution algorithm in more detail. Finally, experimental results are presented and related work is discussed.

2 Example: AbsMinus

In this paper, we are interested in the verification of imperative programs whose semantics has been formally defined within a theorem prover. Our prototype takes as input a small subset of Java that matches our reference semantics³.

³ The Java parser is built using the Eclipse JDT (Java Development Tool)

```

1 class AbsMinus {
2   /*@ ensures   ((i < j) ==> (\result == j-i))
3               && ((i >= j) ==> (\result == i-j));  @*/
4   int absMinus (int i, int j) {
5     int result;
6     int k = 0;
7     if (i <= j) k = k+1;
8     if (k == 1 && i != j) result = j-i;
9     else result = i-j; // ERROR: result = j-i;
10    return result;}}

```

Fig. 1. AbsMinus program in Java

Consider the Java class `AbsMinus` in Figure 1. This computes the absolute value of the subtraction of inputs `i` and `j`. Lines 2-3 give its specification in JML (Java Modelling Language [12]), where `ensures` is the postcondition and `\result` represents the value returned by the program. The JML specification of `AbsMinus` is first parsed into a Hoare triple (see Figure 2). A precondition `p`, a program `c` and a relational postcondition `r` are arguments to a predicate `RSPEC` that specifies the meaning of the Hoare triple:

$$\vdash \forall p \ c \ r. \text{RSPEC } p \ c \ r = \forall s_1 \ s_2. p \ s_1 \wedge \text{EVAL } c \ s_1 \ s_2 \Rightarrow r \ s_1 \ s_2$$

`EVAL c s1 s2` is true if executing `c` in an initial state `s1` results in a final state `s2` (see Section 3.1).

The JML precondition is parsed to a predicate on the initial state, represented by a λ -expression. Since there is no explicit precondition in `AbsMinus`, the parser returns the always-true predicate $(\lambda. \text{T})$. The program is represented using conventional abstract syntax constructors: `Seq` is the sequential execution of instructions, `Cond` is the conditional and `Skip` is the null instruction. The meaning of these constructors are specified in the definition of `EVAL` given in Appendix 0. The postcondition is a relation between the initial and final states, represented as “ $\lambda s_1 \ s_2. \dots$ ”. Such VDM-style relational postconditions reduce the need for ghost (auxiliary) variables and can neatly represent JML’s `\old` construct. States are finite maps from strings to values. Values may be scalars (currently just integers) or arrays (finite maps from indexing numbers to values). Details concerning arrays are omitted from this paper. We will write finite maps in the form $[x_1 \mapsto v_1; \dots; x_n \mapsto v_n]$ and the result of looking up a variable x in a map m as $m \hat{x}$ (the actual notations used in the theorem prover are more cumbersome as they explicitly indicate whether values are scalars or arrays). The symbolic execution of `AbsMinus` proceeds as follows. For each step, we specify if the Theorem Prover alone [*TP*] or both Theorem Prover and External Solvers [*TP-ES*] are used.

A. Compute an initial symbolic state:

$$["i" \mapsto i; "j" \mapsto j; "k" \mapsto k; "Result" \mapsto Result; "result" \mapsto result]$$

In this initial state, $i, j, k, Result$ and $result$ are integer variables, representing the symbolic value in the state of the strings (corresponding to program variables) with the same name. $Result$ is the value returned by the program

```

1 RSPEC
2 (λs. T)
3 (Seq (Assign "result" (Const 0))
4       (Seq (Assign "k" (Const 0))
5             (Seq (Cond
6                   (LessEq (Var "i") (Var "j"))
7                   (Assign "k" (Plus (Var "k") (Const 1)))
8                   Skip)
9             (Seq (Cond
10                  (And (Equal (Var "k") (Const 1))
11                        (Not (Equal (Var "i") (Var "j"))))
12                  (Assign "result" (Sub (Var "j") (Var "i")))
13                  (Assign "result" (Sub (Var "i") (Var "j")))
14                  (Assign "Result" (Var "result"))))))))
15 (λs1 s2. (s1i < s1j ==> s2Result = s1j - s1i) ∧
16          (s1i ≥ s1j ==> s2Result = s1i - s1j))

```

Fig. 2. Relational specification of AbsMinus program

and represents both `\result` of the JML specification and the expression returned by the Java program.

- B. *[TP]* Evaluate the precondition on this initial state by β -reducing the application of the λ -expression on the initial state: \mathbf{T} .
- C. Start with an initial path that contains no condition and thus is equal to \mathbf{T} .
- D. Symbolically execute a path in the program
 - (a) *[TP]* The first instruction (line 3, Fig. 2) is `(Assign "result" (Const 0))`. The new state is computed by mechanised reduction of a small-step semantics that has been formally verified (by interactive proof) to correspond to the big-step reference semantics (see Section 3.1). The new state is: $[i \mapsto i; j \mapsto j; k \mapsto k; \text{Result} \mapsto \text{Result}; \text{result} \mapsto 0]$
 - (b) *[TP]* The next instruction (line 4) is executed in the same way to get: $[i \mapsto i; j \mapsto j; k \mapsto 0; \text{Result} \mapsto \text{Result}; \text{result} \mapsto 0]$
 - (c) *[TP-ES]* The next instruction (line 5) is a conditional. Our symbolic execution algorithm combines the theorem prover and some external solvers to test if the path is feasible (see Section 4.2). The condition is first evaluated on the current state, by reduction of the formal semantics of Boolean operations defined for the language. The result is: $i \leq j$. We then test if this condition is possible according to the precondition (which is \mathbf{T}) and the current path (which is also \mathbf{T}). The SMT solver is called and trivially finds solution $(i,0)$ $(j,0)$. So condition $i \leq j$ is added into the current path and symbolic execution continues on the ‘then’ part (line 7).
 - (d) *[TP]* `(Assign "k" (Plus (Var "k") (Const 1)))` is executed to get: $[i \mapsto i; j \mapsto j; k \mapsto 1; \text{Result} \mapsto \text{Result}; \text{result} \mapsto 0]$
 - (e) *[TP-ES]* The next instruction (line 9) is a conditional branching on: `(And (Equal (Var "k") (Const 1)) (Not (Equal (Var "i") (Var "j"))))`. This is evaluated on the current state to $\neg(i = j)$ using the semantics of Boolean expressions. The SMT solver is then called to test if this condition is possible on the current path $i \leq j$ and it trivially finds solu-

```

if i <= j then
  if ¬(i = j) then
    RESULT["i"↦i; "j"↦j; "k"↦1; "Result"↦j-i; "result"↦j-i]
  else RESULT["i"↦i; "j"↦j; "k"↦1; "Result"↦i-j; "result"↦i-j]
else RESULT["i"↦i; "j"↦j; "k"↦0; "Result"↦i-j; "result"↦i-j]

```

Fig. 3. Result of symbolic execution of `AbsMinus`

tion $(i,-1) (j,0)$. So $\neg i = j$ is added into the current path and execution continues on the ‘then’ part (line 12).

(f) *[TP]* The two last instructions on the path (line 12 and line 14) are executed and the end of the path $i \leq j \wedge \neg(i=j)$ is reached with state:

$["i" \mapsto i; "j" \mapsto j; "k" \mapsto 1; "Result" \mapsto j-i; "result" \mapsto j-i]$.

E. *[TP]* Now the postcondition relation between the initial and final states is computed by β -reducing the application of the λ -expression (lines 15-16), resulting in: $i \geq j \implies (j - i = i - j)$ ⁴.

We then show that the precondition and the path imply the postcondition:

$(i \leq j \wedge \neg(i=j)) \implies (i \geq j \implies (j - i = i - j))$.

This is easily proved using simplification in the theorem prover (see part 4.3).

F. The next step is to backtrack to the previous conditional instruction to explore another path in the program. Execution goes to step D.(e) and tests if the negation of the condition is possible. Since it is possible, execution continues on the ‘else’ part which gives a correct path.

G. The next backtrack goes to step D.(c). Condition $\neg(i \leq j)$ is possible so symbolic execution continues on the `skip` instruction (line 8), which does not modify the current state. Then the conditional on line 9 is reached. Since the value of “k” in the current state is 0, the condition:

$(\text{And } (\text{Equal } (\text{Var } "k") (\text{Const } 1)) (\text{Not } (\text{Equal } (\text{Var } "i") (\text{Var } "j"))))$

is evaluated by the theorem prover to false and so only the ‘else’ part is explored. This again gives a correct path.

The symbolic execution returns a conditional term that represents the paths that have been successively explored. Each possible value of this term is an outcome which gives the final value of the state. This outcome is preceded by `RESULT` when the path is correct, and by `ERROR` when the path contains an error or `TIMEOUT` if symbolic execution failed to reach the end of the program for the given number of steps (see figure 5). Figure 3 shows the result of the symbolic execution of `AbsMinus` program.

AbsMinus program with an error

We now consider another version of `AbsMinus` where a ‘copy-paste’ error results in $j-i$ being returned instead of $i-j$ (see line 9 in Figure 1).

When taking path $i > j$ in the program, the symbolic execution ends with state:

$["i" \mapsto i; "j" \mapsto j; "k" \mapsto 1; "Result" \mapsto j-i; "result" \mapsto j-i]$

⁴ Note that the first part of the conjunction in the lambda expression has been evaluated to true because $Result = j-i$

```

if i <= j then
  if ¬(i = j) then
    RESULT["i"↦i; "j"↦j; "k"↦1; "Result"↦j-i; "result"↦j-i]
  else RESULT["i"↦i; "j"↦j; "k"↦1; "Result"↦j-i; "result"↦j-i]
else ERROR["k"↦0; "Result"↦j-i; "result"↦j-i; "i"↦(1); "j"↦(0)]

```

Fig. 4. Result of symbolic execution of AbsMinus program with an error

The relational postcondition is then evaluated to $i >= j \implies (j - i = i - j)$ but the theorem prover fails to show that $i > j \implies (i >= j \implies (j - i = i - j))$. So the SMT solver is called and it trivially finds that solution $(i, 1)$ and $(j, 0)$ satisfies the negation of this formula. This provides a counterexample to the correctness of the program. Figure 4. shows the result of the symbolic execution of AbsMinus with this error.⁵

3 Infrastructure provided by the theorem prover

In this section we present the infrastructure provided by the theorem prover. The formal definition of the operational semantics of the imperative language is in Appendix 0. Our prototype uses HOL4, but the same methodology could be implemented with other theorem provers.

3.1 Operational semantics

We represent programs and specifications as terms in higher order logic. The formal representation of programs shown in Figure 2 has a standard big-step operational semantics [4] shown in Appendix 0.

The big-step semantics is not easy to execute directly because it is an inductive relation⁶, so we defined a small-step semantic function `STEP1` (see Appendix 0). If `STEP1(l1, s1) = (l2, r)` then executing one step of the command at the head of `l1` in state `s1` results in `(l2, r)`, where `l2` are the remaining commands to be executed and `r` is the result, which can either be `RESULT(s2)` if the step succeeds or `ERROR(s2)` if there is an assertion failure. There is a third kind of result, `TIMEOUT(s2)`, which is not generated by `STEP1` but can arise when executing sequence of steps (see figure 5).

It is a straightforward mechanical proof (which we have done using HOL4) that the small-step semantic function corresponds to `EVAL` [15]. Define a small-step transition relation by:

$$\text{SMALL_EVAL } (l1, s1) (l2, s2) = (\text{STEP1 } (l1, s1) = (l2, \text{RESULT } s2))$$

where `l1` and `l2` are lists of commands (`[]` is the empty list and `[c]` is the singleton list containing `c`). A routine mechanical proof then establishes:

$$\vdash \forall c \ s1 \ s2. \text{ EVAL } c \ s1 \ s2 = \text{TC SMALL_EVAL } ([c], s1) ([] , s2)$$

where `TC SMALL_EVAL` is the transitive closure of `SMALL_EVAL`

⁵ Note that the conditional term contains two correct paths and one path with an error.

⁶ Inductive relations can be executed using techniques adapted from Prolog interpreters, but we feel our approach meshes more naturally with BMC.

3.2 Execution of the semantics within the theorem prover

We use mechanised proof to compute symbolic single steps and conditions when executing a path. Functions `STEP1` and `beval` (see Appendix 0) are efficiently executed inside HOL4 using a call-by-value reduction engine due to Barras [3]. More precisely, we have implemented three functions (in Standard ML, the HOL4 met-language): `nextState c s` that returns the next state after executing commands `c` on state `s`, if the outcome is a `RESULT` and raises an exception otherwise, `nextInst c s` which returns the next list of instructions to execute and `evalCond b s` which returns the value of term `b` on state `s`. Both `nextState c s` and `nextInst c s` execute `STEP1` using the call-by-value reduction engine while `evalCond b s` executes `beval`. For example, for `AbsMinus` program (see Section 2), when `b` is the term `(And (Equal (Var "k") (Const 1)) (Not (Equal (Var "i") (Var "j"))))` and state `s` is the finite map `["i" ↦ i; "j" ↦ j; "k" ↦ 1; "Result" ↦ Result; "result" ↦ 0]`, `evalCond b s` executes `beval b s` using the reduction engine to get theorem: $\vdash \text{beval } b \ s = (\text{Not } (\text{Equal } (\text{Var } "i") (\text{Var } "j")))$, and returns the right-hand side of the conclusion of the theorem.

3.3 Simplification tools

We also have implemented in HOL4 formal rewriting and custom derived rules for transformations on pre and postconditions. For example, converting `\forall` JML statements to finite conjunctions, e.g. the precondition of `Bsearch` program (see Appendix 3) `\forall int i; (i >= 0 && i < a.length - 1); a[i] <= a[i+1]` is converted to the conjunction $a_0 \leq a_1 \wedge \dots \wedge a_8 \leq a_9$ when the length of the array has been fixed to 10. When necessary, pre and postconditions are also transformed to a form suitable to submitting to a constraint solver (this was performed by unverified Java programs in earlier work [7]). For example, formulae of the form $\neg((A_1 \Rightarrow B_1) \wedge \dots \wedge (A_n \Rightarrow B_n) \wedge t)$ are converted to $(A_1 \wedge \neg B_1) \vee \dots \vee (A_n \wedge \neg B_n) \vee \neg t$, and the several terms of the conjunction are solved in sequence by the constraint solver. The implementation of such truth preserving term manipulations is straightforward, so is not elaborated here.

4 Integrating theorem prover and external solvers

Before presenting the algorithm of symbolic execution in more detail, we first explain how our theorem prover and external solvers can be integrated and used to test the feasibility or correctness of program execution paths.

We use two external solvers: the SMT solver `yices` (see [18], [9]) that we selected for its efficiency but also its simple input format. Since `yices` doesn't handle non-linear expressions, we also use the constraint solver `JSOLVER` which is a version of the commercial `Ilog` solver which has been developed for verifying properties of business rules (see [13] for main features of `JSOLVER`).

4.1 Calling external solvers from the HOL4 theorem prover.

The HOL4 system provides many predefined functions for proving theorems (e.g. various decision procedures and basic first-order resolution solvers). We defined two SML functions: `extSMTSolV tm` to invoke an external SMT solver

and `extCPSolv tm to f` to invoke an external constraint solver. The first argument, `tm`, is an existentially quantified term whose satisfiability is to be checked, the second argument, `to`, is a timeout. A third argument `f` is necessary when calling the constraint solver to set the domains of variables in the constraint system. Function `extSMTSolv tm to` is called when `tm` is linear and `extCPSolv tm to f` when `tm` is non-linear.

Invoking one of these functions returns a *tagged theorem*, where the tag (which is propagated whenever the theorem is used) indicates how the theorem was proved. The tagged theorem is built as follows:

- If the external solver does not find any solution, then the theorem $\vdash(\text{tm}=\text{F})$ is returned. This theorem is tagged with the string `SMTSolver` when the SMT solver is used and tagged with the string `CPSolver:f` when the constraint solver is used. This means that external constraint solver has shown that there exists no value in $[-2^{f-1}, 2^{f-1} - 1]$ that satisfies `tm` and thus that $\neg\text{tm}$ is true for integers coded on `f` bits,
- If the external solver finds a solution, then this solution is used to instantiate the existentially quantified variables in `tm` and the theorem $\vdash(\text{tm}=\text{T})$ is proved. This theorem is not tagged as externally generated since its proof has been done inside HOL4 using the witness from the solver.

4.2 Testing feasibility of paths

A key point of symbolic execution, compared to other BMC methods where a single formula represents all execution paths of a bounded length, is that only semantically feasible paths are explored. But this is an improvement only if feasibility testing is very fast. So external solvers are employed to boost efficiency.

Let *test* be the Boolean expression occurring in a conditional branch. The following two-stage method is used to decide feasibility of *test*:

1. Evaluate *test* on the current state from the semantics of Boolean expressions using `evalCond` function (see 3.2). Let *simpTest* be the result. If *simpTest* is T (true) or F (false) then return the corresponding value and stop.
2. Let $\Phi = \text{pre} \wedge \text{path} \wedge \text{simpTest}$ where *pre* is the precondition which has been evaluated on the initial state, and *path* is the current path (i.e conjunction of decisions that have been taken so far). Call the external solver to test satisfiability of Φ with a small timeout (and a small integer format when the constraint solver is used).
 - (a) If there is a solution, then return true,
 - (b) If there is no solution, or a timeout, use HOL4's simplification and integer decision procedures to try to decide the satisfiability of Φ .

Symbolic evaluation (1) gives the assurance that the term to be tested conforms to the formal semantics of the language. Calling the external solver (2) improves efficiency. Step (2) is either a call to an SMT solver on a small formula, or either a call to the constraint solver on a small formula and a small domain⁷. Note that even when an external solver has been used, assurance is still preserved since the theorem prover is called if the external solver has shown that the path is infeasible.

⁷ if the path is possible for a value inside $[-2^{f-1}, 2^{f-1} - 1]$ where *f* is small (e.g *f* = 8) it is a fortiori true for a larger domain

4.3 Testing correctness of paths

While efficiency is the key point when testing feasibility of paths, soundness is a priority when testing their correctness. So the theorem prover is called first and only if that fails is the external solver called using a large timeout (and large integer format with the constraint solver). If the solver succeeds then a tagged theorem is returned. If the timeout is reached, or if higher assurance than that provided by the solver is required, then interactive proof can be done.

5 Symbolic execution

Symbolic execution is by depth first search of feasible paths. A user-specified parameter *count* bounds the number of steps (e.g. when programs contain loops). Let *pre*, *path*, *s*₁, *s*₂, *post* be terms as follows:

- *pre*: precondition (predicate on states represented as a λ -expression)
- *path*: current path (conjunction of decisions taken so far)
- *s*₁: initial state before program execution
- *s*₂: current state after execution of the current path
- *post*: postcondition (predicate on pairs of states represented as a λ -expression)

The initial state *s*₁ is automatically built from the program, with logical variables representing the symbolic values of the program variables. Let **1** be the list of terms that represent the instructions of the program (initially this will be [c], where c is the program being symbolically executed). Let *valPre* be the precondition evaluated on *s*₁, and *valPost* be the postcondition evaluated on the pair (*s*₁, *s*₂). We assume that we have two functions:

- **testPath** tests if condition *b* is feasible on the current path
- **verifyPath** tests the correctness of the path i.e if $valPre \wedge path \wedge \neg valPost$ has no solution⁸. This function returns the outcome (**RESULT** *s*) if the program is correct along the path and otherwise returns the outcome (**ERROR** *s_{err}*) where *s_{err}* contains the error that has been found.

These two functions call the external solvers and the theorem prover as explained in section 4.2 and 4.3.

The symbolic execution algorithm is detailed in Figure 5. If the last instruction has been reached (point 1) then the correctness of the path is tested. If the maximum number of steps (*count*) reaches zero (point 3) then an outcome **TIMEOUT**(*s*) is generated. If the first instruction is not a conditional (point 5), then next state and next instruction list is computed using the small-step semantics. If the first instruction is a conditional instruction (point 6) then the feasibility of the condition and the feasibility of the negation of the condition are tested. Note that backtracking is performed when the instruction is a conditional instruction or a **While**-instruction because the two possible paths are

⁸ this means that $valPre \wedge path \wedge \neg valPost$ is false for each input value and so that $valPre \wedge path \Rightarrow valPost$ is true

```

execSymb(pre, path, l, s1, s2, count, post) =
1. If  $l=[]$  the end of a path is reached so result is verifyPath(pre, path, s1, s2, post)
2. else
3. if count = 0 then the program can't be executed with the given number
of execution steps, so the result is (TIMEOUT s2).
4. else let  $l = [c, l']$ 
5. if c is not a control instruction (Cond or While) then let
 $s' = \text{nextState}(l, s_2)$  and  $l'' = \text{nextInst } l \ s_2$ . Recursively call
execSymb(pre, path, l'', s1, s', count-1, post) to continue symbolic execution.
6. else let b be the condition of c, then call testPath(pre, s2, path, b)
to know if the condition is possible on the current path
7. if b is possible, take the corresponding path in the program.
8. If c is the conditional (Cond b cthen celse) then
recursively call execSymb(pre, (path ∧ b), [cthen, l'], s1, s2, count, post)
to execute the then part.
9. If c is the loop instruction (While b cwhile) then recursively call
execSymb(pre, (path ∧ b), [cwhile, (While b cwhile), l'], s1, s2, count, post)
to enter the loop.
10. if b is not possible, take the corresponding path in the program.
11. If c is the conditional (Cond b cthen celse) then
recursively call execSymb(pre, (path ∧ ¬b), [celse, l'], s1, s2, count, post)
to execute the else part.
12. If c is the loop instruction (While b cwhile) then recursively call
execSymb(pre, (path ∧ ¬b), l', s1, s2, count, post)
to exit the loop.
13. Call testPath(pre, s2, path, ¬b) to know if the
negation of the condition is possible on the current path and take
the corresponding path as explained above

```

Fig. 5. Symbolic execution algorithm

explored (i.e points 6 and 13 are both executed). For While-instructions, this backtracking implies that all feasible paths through the loop that require less than *count* execution steps are tested. As a consequence, the conditional term returned by the symbolic execution may contain RESULT outcomes for paths that exit the loop with less than *count* steps, and TIMEOUT outcomes for other cases.

6 Experimental results

In this section, we report experimental results for a set of textbook algorithms. All experiments were performed on an Intel(R) Pentium(R) M processor 1.86GHz with 1.5G of memory. Table 1 shows statistics on the solving process for our set of examples. In this table, #cond (resp. #const and #cut) is the number of conditions reached in the program (resp. that have been reduced to a constant by `evalCond` and that have been proven false). #path is the number of feasible paths. #solveHOL is the number of paths that have been verified using HOL4. HOLtime (resp. solverTime) is the time spent by HOL4 for proofs and symbolic execution (resp. with the external SMT and constraint solvers). For Sum and SumPtoN, we verified all numbers of loop steps from 0 to 10, and for Bsearch and BubbleSort program, the length of the array has been set to 10.

6.1 Set of programs

Tritype Our first example is not a textbook algorithm, however we selected it because it illustrates programs that do not contain loops but have complex conditional statements and numerous non-feasible paths. This kind of control structure is frequently found in command and control systems. This program takes three positive integers as inputs (the triangle sides) and returns a value that determines the type of the triangle (see Appendix 1).

As shown in Table 1, all paths were solved using HOL4. Furthermore, half of the conditions were decided by `evalCond` because many tests in this program are on a local variable which takes constant values, according to the number of equal sides.

Sum of the n first integers computes the sum of the n first integers. The specification is that it returns $n \times (n + 1)/2$ (where n is the data input).

The symbolic execution sets a constant value for n because it adds entrance and exit conditions of the loops (see points 9 and 12 in figure 5). For example, when the loop has been entered 5 times, the term to be verified is:

$\exists n.(n \geq 0) \wedge (0 \leq n \wedge 1 \leq n \wedge 2 \leq n \wedge 3 \leq n \wedge 4 \leq n \wedge \neg(5 \leq n)) \wedge \neg(10 = n * (n + 1)/2)$. Since n must satisfy both $4 \leq n$ and $\neg(5 \leq n)$ it follows that $n = 4$, so the postcondition $\neg(10 = n * (n + 1)/2)$ trivially holds.

The results in Table 1 show that 11 paths have been explored (entering 0, 1, 2, ..., 10 times into the loop) and that all the paths were solved with the constraint solver. This is because simplification in HOL4 doesn't handle non linear terms. However, since the value of n is constant as explained above, the constraint solver efficiently verifies this term.

Sum of integers from P to N computes the sum of the integers from p to n where p and n are input data (see Appendix 2). The precondition is that p is less or equal to n and the postcondition is that the result is $n \times (n + 1)/2 - (p - 1) \times p/2$. This example illustrates a non-linear case. The sum starts with input data p , so the decisions taken to enter the loop, or not, won't set the value of n , since it will depend on p .

The results in Table 1 show that execution time for HOL4 is 20 times slower than for the `Sum` program, because simplification rules take more time trying to simplify the term before they finally fail. Also, the constraint solver takes 400 more times because here the term to be verified is not instantiated.

Binary search is the well known binary search program that determines if a value x is present in a sorted array a . We also consider an incorrect version of this program where a copy-paste error has been inserted (see Appendix 3).

The results in Table 1 show that all paths of the `Bsearch` program were solved with HOL4. The SMT solver found two errors in the incorrect `Bsearch` program. The first one for path $\neg(a_4 = x) \wedge x < a_4 \wedge \neg(a_1 = x) \wedge \neg(x < a_1)$ and data input $x = 1$ and $a = [0, 0, 0, 1, 2, 2, 2, 2, 2, 2]$.

Bubble sort with precondition is a bubble sort algorithm where a precondition sets the values of the array to be sorted in decreasing order and to contain

	conditions			solved paths		HOLTime	solverTime	
	#cond	#const	#cut	#path	#solveHOL		SMT	CS
Tritype	27	15	16	10	10	39.410s	0.064s	0s
Sum	12	0	1	11	0	17.433s	0.052s	0.032s
SumPtoN	12	0	1	11	0	344.878s	0.036s	13.279s
Bsearch	51	31	21	21	21	237.071s	0.108s	0s
BsearchKO	25	13	13	7	7	119.499s	0.088s	0s
BubbleSort	109	109	10	1	1	526.989s	0.2s	0s

Table 1. Experimental results

the values from $a.length - 1$ to 0 (see Appendix 4). This example is from Mantovani et al. [2].

Table 1 shows that we explored the unique feasible path in this program, and that all conditions were decided by `evalCond` because the precondition sets initial values to the array. The ten conditions that were cut correspond to the ten times the inner loop was exited.

6.2 Discussion

The experimental results have shown that our approach is feasible. All examples, except non-linear ones, were verified with HOL4 and most of path conditions were decided with HOL4. However, if the mechanical reductions performed by the theorem prover increase assurance, they also have a strong effect on performance. Executing the semantics (i.e functions `nextState`, `nextInst` and `evalCond`) is quite fast but may become slow if the size of the state increases (e.g more variables or larger terms). Execution time for `evalCond` varies from 0.054s for `Tritype` to 3.576s for `BubbleSort`, and execution time for `nextState` varies from 0.058s for `Sum` to 5.336s for `BubbleSort`. Function `nextState` is called very often and thus it has many impact on performance. Furthermore, HOL4 decision procedures used to test correctness at the end of the paths are rarely called but are very time consuming, from 3.340s for `Tritype` to 96.846s for `BubbleSort`.

7 Related work

The symbolic execution algorithm presented here is inspired by previous work on BMC within a constraint programming framework (see [6, 7] and [11] for the original representation of imperative programs as constraint systems). In [7], constraint-based solvers⁹ were used for pruning infeasible paths and solving conditions at the end of the paths. Performance of the approach presented in this paper are in average two hundred times slower.

The alternative BMC method for software verification consists in building a formula whose models correspond to program execution paths of bounded length that violate a property. The satisfiability of this formula is checked by replacing

⁹ More precisely, this previous approach combined the ILOG CPLEX MIP Mixed Integer Programming tool which is based on the simplex algorithm with ILOG JSOLVER we used in this paper

arithmetic operators by bit-vectors operators to obtain a propositional formula which is solved using efficient SAT solvers.

CBMC [14, 5] is one of the most popular tools that implements BMC for checking C programs. CBMC supports major features of ANSI C such as pointers or dynamic memory, and also handles preconditions and assertions. CBMC is more efficient than our tool (e.g. 30 times faster for `Tritype` program and 10 times faster for `Bsearch`). However, CBMC also has some limitation. Some additional information is required in CBMC for proving the `Tritype` program to ensure that there is no overflow into the sums in modular arithmetic, and CBMC is not able to verify `Bsearch` program for arrays of length greater or equal to 32 (timeout of one hour)¹⁰.

In [2], Armando et al. generalised the CBMC approach by encoding the programs into a quantifier free formula to be checked for satisfiability using a SMT solver. In [1], they lifted their model checking procedure for linear programs to deal with arrays via iterative abstraction refinement. This approach was implemented in the `EUREKA` tool, and applied to text book algorithms (gray code, sorting algorithms). However, reference [1] reports that the `EUREKA` tool verified the `BubbleSort` program in 91.92s for an array of length 8 and that this is the greatest instance that can be proved with a timeout of $30mn$ (while it takes $16mn$ with our approach for an array of length 10).

It is also interesting to contrast our work to some model-checking approaches that perform symbolic execution along a path in the control flow graph (see [16, 17]). Pasareanu and Visse described in [16] a framework based on symbolic execution algorithm and loop invariant discovering for the verification of Java programs. The symbolic execution algorithm we use is very similar to theirs since they also perform a forward symbolic execution using decision procedures for testing feasibility of paths. However, they aim to perform a complete verification (by discovering or providing invariants), while we do a bounded verification. The main contribution of our symbolic execution algorithm is that it is automatically derived from a semantics that has been formally defined within a theorem prover.

Another interesting piece of work, that combines theorem proving with automatic solvers as we do in this paper, is presented in [10]. The `Why/Krakatoa/Caduceus` platform is based on the `Why` language, which is dedicated to program verification. Both Java and C programs are translated to `Why` programs. A tool based on a Weakest Precondition calculus generates verification conditions for interactive provers, such as `Coq` or `Isabelle` and automatic provers such as `Simplify` or SMT solvers `Yices`. This approach differs from ours because it does a complete partial verification, and thus may require that invariants are given, while we do a fully automatic bounded verification.

8 Future work

We mentioned in the introduction that the work reported here lies in the middle of a verification tool implementation spectrum with unverified tools at one end and everything computed by deduction inside a theorem prover at the other

¹⁰ experiments were performed with version 2.8 of CBMC

end. In the future we plan to explore other points in this spectrum. In particular we would like to perform the complete symbolic execution and path extraction within a theorem prover and compare performance and scalability with the approach described here. Preliminary experiments suggest that this is feasible, but there are efficiency challenges. However, in applications requiring certification, having high assurance that verification is sound is valuable, so it is useful to calibrate the cost of increasing soundness assurance, even if for some applications the price is too high.

References

1. Armando A., Benerecetti M., and Mantovani J. Abstraction Refinement of Linear Programs with Arrays. Proceedings of TACAS 2007, LNCS 4424: 373-388, 2007.
2. 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: 146-162, 2006
3. Bruno Barras. Programming and Computing in HOL. Theorem Proving in Higher Order Logics, TPHOLS'2000, LNCS 1869: 17-37, 2000.
4. Juanito Camilleri and Tom Melham, Reasoning with Inductively Defined Relations in the HOL Theorem Prover, Technical Report 265, Computer Laboratory, University of Cambridge, 1992. <http://www.comlab.ox.ac.uk/tom.melham/pub/Camilleri-1992-RID.pdf>
5. Clarke E., Kroening D., Lerda F. A Tool for Checking ANSI-C programs. Procs of TACAS 2004, LNCS 2988: 168-176, 2004
6. Collavizza H. and Rueher M. Software Verification using Constraint Programming Techniques. Procs of TACAS 2006, LNCS 3920: 182-196, 2006.
7. Collavizza H., Rueher M., and van Hentenryck P. A Constraint-Programming Framework for Bounded Program Verification. Proc. of Constraint Programming Conference CP'2008, LNCS 5202: 327-341.
8. Vijay D'Silva, Daniel Kroening and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 27, N 7, July 2008.
9. Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). CAV 2006, LNCS 4144: 81-94.
10. Filliâtre J.C., Claude Marché: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification Proc. CAV'2007, LNCS 4590: 173-177, 2007.
11. Gotlieb A., Botella B. and Rueher M: Automatic Test Data Generation using Constraint Solving Techniques. Proc. ISSTA 98, ACM SIGSOFT (2), 1998.
12. JML home page <http://www.cs.ucf.edu/~leavens/JML/>
13. Michel Leconte, Bruno Berstel: Extending a CP solver with congruences as domains for program verification. Workshop on Software Testing, Verification and Analysis, Constraint Programming conference, 2006.
14. D. Kroening, E. M. Clarke, and K. Yorav, Behavioral consistency of C and Verilog programs using bounded model checking, in Proc. DAC, 2003, pp. 368-371.
15. Tobias Nipkow. Winkler is (almost) Right: Towards a Mechanized Semantics Textbook. In Foundations of Software Technology and Theoretical Computer Science, LNCS 1180, 1996, 180-192.
16. Corina S. Pasareanu, Willem Visser: Verification of Java Programs Using Symbolic Execution and Invariant Generation. SPIN 2004: 164-181
17. W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda: Model Checking Programs. Automated Software Engineering Journal. Volume 10, Number 2, April 2003.
18. Yices: An SMT Solver. <http://yices.csl.sri.com/index.shtml>

Appendix 0: Semantics

Boolean expressions: $\text{beval } b \ s$ is the value of b in s . Similarly $\text{neval } e \ s$ is the value of expression e in s (details omitted here).

$$\begin{aligned}
& (\forall e1 \ e2 \ s. \text{beval } (\text{Equal } e1 \ e2) \ s = (\text{neval } e1 \ s = \text{neval } e2 \ s)) \wedge \\
& (\forall e1 \ e2 \ s. \text{beval } (\text{Less } e1 \ e2) \ s = (\text{neval } e1 \ s) < (\text{neval } e2 \ s)) \wedge \\
& (\forall e1 \ e2 \ s. \text{beval } (\text{LessEq } e1 \ e2) \ s = (\text{neval } e1 \ s) <= (\text{neval } e2 \ s)) \wedge \\
& (\forall b1 \ b2 \ s. \text{beval } (\text{And } b1 \ b2) \ s = (\text{beval } b1 \ s \wedge \text{beval } b2 \ s)) \wedge \\
& (\forall b1 \ b2 \ s. \text{beval } (\text{Or } b1 \ b2) \ s = (\text{beval } b1 \ s \vee \text{beval } b2 \ s)) \wedge \\
& (\forall b \ s. \text{beval } (\text{Not } b) \ s = \sim(\text{beval } b \ s))
\end{aligned}$$

Big-step semantics of commands: $s+(v,n)$ is the state obtained from s by making variable v have value n ; $s^{\wedge}v$ is the value of v in s , $v \in s$ means v is defined in s and $s-v$ is the result of removing v from s .

$$\begin{aligned}
& (\forall s. \text{EVAL Skip } s \ s) \\
& \wedge (\forall s \ v \ e. \text{EVAL } (\text{Assign } v \ e) \ s \ (s+(v,(\text{neval } e \ s)))) \\
& \wedge (\forall s \ v. \text{EVAL } (\text{Dispose } v) \ s \ (s-v)) \\
& \wedge (\forall c1 \ c2 \ s1 \ s2 \ s3. \text{EVAL } c1 \ s1 \ s2 \wedge \text{EVAL } c2 \ s2 \ s3 \\
& \quad \Rightarrow \text{EVAL } (\text{Seq } c1 \ c2) \ s1 \ s3) \\
& \wedge (\forall c1 \ c2 \ s1 \ s2 \ b. \text{EVAL } c1 \ s1 \ s2 \wedge \text{beval } b \ s1 \\
& \quad \Rightarrow \text{EVAL } (\text{Cond } b \ c1 \ c2) \ s1 \ s2) \\
& \wedge (\forall c1 \ c2 \ s1 \ s2 \ b. \text{EVAL } c2 \ s1 \ s2 \wedge \neg(\text{beval } b \ s1) \\
& \quad \Rightarrow \text{EVAL } (\text{Cond } b \ c1 \ c2) \ s1 \ s2) \\
& \wedge (\forall c \ s \ b. \neg \text{beval } b \ s \Rightarrow \text{EVAL } (\text{While } b \ c) \ s \ s) \\
& \wedge (\forall c \ s1 \ s2 \ s3 \ b. \\
& \quad \text{EVAL } c \ s1 \ s2 \wedge \text{EVAL } (\text{While } b \ c) \ s2 \ s3 \wedge \text{beval } b \ s1 \\
& \quad \Rightarrow \text{EVAL } (\text{While } b \ c) \ s1 \ s3) \\
& \wedge (\forall c \ s1 \ s2 \ v. \\
& \quad \text{EVAL } c \ s1 \ s2 \\
& \quad \Rightarrow \text{EVAL } (\text{Local } v \ c) \ s1 \ (\text{if } v \in s1 \ \text{then } s2+(v,(s1^{\wedge}v)) \ \text{else } s2-v))
\end{aligned}$$

Small-step semantics of commands: $::$ is the list ‘cons’ operation.

$$\begin{aligned}
& (\text{STEP1}([], s) = ([], \text{ERROR } s)) \\
& \wedge (\text{STEP1}(\text{Skip } :: 1, s) = (1, \text{RESULT } s)) \\
& \wedge (\text{STEP1}(\text{Assign } v \ e :: 1, s) = (1, \text{RESULT}(s+(v,(\text{neval } e \ s)))))) \\
& \wedge (\text{STEP1}(\text{Dispose } v :: 1, s) = (1, \text{RESULT}(s-v))) \\
& \wedge (\text{STEP1}(\text{Seq } c1 \ c2 :: 1, s) = (c1 :: c2 :: 1, \text{RESULT}(s))) \\
& \wedge (\text{STEP1}(\text{Cond } b \ c1 \ c2 :: 1, s) = \\
& \quad \text{if } \text{beval } b \ s \ \text{then } (c1 :: 1, \text{RESULT } s) \ \text{else } (c2 :: 1, \text{RESULT } s)) \\
& \wedge (\text{STEP1}(\text{While } b \ c :: 1, s) = \\
& \quad \text{if } \text{beval } b \ s \ \text{then } (c :: \text{While } b \ c :: 1, \text{RESULT } s) \\
& \quad \quad \text{else } (1, \text{RESULT } s)) \\
& \wedge (\text{STEP1}(\text{Local } v \ c :: 1, s) = \\
& \quad \text{if } v \in s \ \text{then } (c :: \text{Assign } v \ (\text{Const}(s^{\wedge}v)) :: 1, \text{RESULT } s) \\
& \quad \quad \text{else } (c :: \text{Dispose } v :: 1, \text{RESULT } s))
\end{aligned}$$

Appendix 1: Tritype program

```

/** program for triangle classification
 * @return 1 if (i,j,k) are the sides of any triangle
 *         2 if (i,j,k) are the sides of an isosceles triangle
 *         3 if (i,j,k) are the sides of an equilateral triangle
 *         4 if (i,j,k) are not the sides of any triangle */
class Tritype {
    /*@ 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));
    @*/
    static int tritype (int i, int j, int k) {
        int trityp = 0;
        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) {
                if ((i+j)<=k || ((j+k)<=i || (i+k)<=j))
                    trityp = 4;
                else
                    trityp = 1;
            }
            else {
                if (trityp > 3) trityp = 3;
                else {
                    if (trityp == 1 && (i+j) > k) {
                        trityp = 2;
                    }
                    else {
                        if (trityp == 2 && (i+k) > j) {
                            trityp = 2;
                        }
                        else {
                            if (trityp == 3 && (j+k) > i) {
                                trityp = 2;
                            }
                            else {
                                trityp = 4;
                            }
                        }
                    }
                }
            }
        }
        return trityp;
    }
}

```

Appendix 2: Sum of integers from P to N

```

/* Computes the sum of integers from p to n.
 */
public class SumFromPtoN{
    /*@ requires (n >= 0) && (p >= 0) && (p<=n) ;
    @ ensures \result == n*(n+1)/2 - (p-1)*p/2;
    @*/
    int sum (int p,int n) {
        int i = p;

```

```

int s = 0;
while (i<=n) {
    s = s+i;
    i = i+1;
}
return s;
}
}

```

Appendix 3: Bsearch program

```

class Bsearch {
    /*@ requires (\forallall int i; (i >= 0 && i < a.length -1); a[i]<=a[i+1]);
    @ ensures
    @ ((\result == -1) ==> (\forallall int i; (i >= 0 && i < a.length); a[i] != x))
    @ && ((\result != -1) ==> (a[\result] == x));
    @*/
    int binarySearch (int[] a, int x) {
        int result = -1;
        int mid = 0;
        int left = 0;
        int right = a.length -1;
        while (result == -1 && left<=right) {
            mid = (left + right) / 2;
            if (a[mid] == x)
                result = mid;
            else {
                if (a[mid] > x)
                    right = mid - 1;
                else
                    left = mid + 1; //ERROR: right = mid - 1;
            }
        }
        return result;
    }
}

```

Appendix 4: Bubble Sort program

```

/* Bubble sort with a precondition:
 * the array contains numbers between 0 and a.length
 * sorted in decreasing order
 */
class BubbleSort {
    /*@ requires (\forallall int i; 0<= i && i < a.length; a[i] == (a.length-1)-i);
    @ ensures (\forallall int i; 0<= i && i < a.length-1; a[i]<=a[i+1]);
    @*/
    static void sort(int[] a) {
        int i=0;
        while (i<a.length-1){
            int j=0;
            while (j < (a.length-i)-1) {
                if (a[j]>a[j+1]) {
                    int aux = a[j];
                    a[j]= a[j+1];
                    a[j+1] = aux;
                }
                j=j+1;
            }
            i=i+1;
        }
    }
}

```