Exploration of the Capabilities of Constraint Programming for Software Verification

Hélène Collavizza and Michel Rueher

Université de Nice–Sophia-Antipolis – I3S/CNRS,
930, route des Colles - B.P. 145, 06903 Sophia-Antipolis, France
{helen, rueher}@essi.fr

Abstract. Verification and validation are two of the most critical issues in the software engineering process. Numerous techniques ranging from formal proofs to testing methods have been used during the last years to verify the conformity of a program with its specification. Recently, constraint programming techniques have been used to generate test data. In this paper we investigate the capabilities of constraint programming techniques to verify the conformity of a program with its specification. We introduce here a new approach based on a transformation of both the program and its specification in a constraint system. To establish the conformity we demonstrate that the union of the constraint system derived from the program and the negation of the constraint system derived from its specification is inconsistent (for the considered domains of values). This verification process consists of three steps. First, we generate a Boolean constraint system which captures the information provided by the control flow graph. Then, we use a SAT solver to solve the Boolean constraint system. Finally, for each Boolean solution we build a new constraint system over finite domains and solve it. The latter system captures the operational part of the program and the specification. Boolean constraints play an essential role since they drastically reduce the search space before the search and enumeration processes start. Moreover, in the case where the program is not conforming with its specification, Boolean constraints provide a powerful tool for finding wrong behaviours in different execution paths of the program. First experimental results on standard benchmarks are very promising.

1 Introduction

Verification and validation are two of the most critical issues in the software engineering process. These expensive and difficult tasks may account for up to 50% of the cost of software development [12]. Numerous techniques ranging from formal proofs to testing methods have been used during the last years to verify the conformity of a program with its specification. The goal of the SLAM project is to build "tools that can do actual proofs about the software and how it works in order to guarantee the reliability"[1].

[1] See http://research.microsoft.com/slam

H. Hermanns and J. Palsberg (Eds.): TACAS 2006, LNCS 3920, pp. 182–196, 2006.

Constraint programming techniques have been used to generate test data (e.g., [9, 10, 22, 23]) and to develop efficient model checking tools (e.g. [17, 6]). SAT based model checking platforms have been able to scale and perform well due to many advances in SAT solvers [20]. Recently Bouquet et al [3] developed a symbolic animator for specifications written in Java Modeling Language (JML) [15]. Their JML animator– based on constraint programming techniques– allows to simulate the execution of a JML specification and to verify on the fly class invariant properties.

In this paper we investigate the capabilities of constraint programming techniques to verify the conformity of a program with its specification. We introduce a new approach based on a transformation of both a program and its specification in a constraint system. To establish the conformity we demonstrate that the union of the constraints derived from the program and the negation of the constraints derived from its specification is inconsistent. Roughly speaking, pruning techniques -that reduce the domain of the variables- are combined with search and enumeration heuristics to demonstrate that this constraint system has no solutions.

The verification process consists of three steps:

1. Generating of a Boolean constraint system which captures the information provided by the control flow graph of the program and the specification;
2. Using a SAT solver to find the solutions of the Boolean constraint system. For each Boolean solution a new constraint system over finite domains – denoted CSP in the following– is built; the latter captures the operational part of the program and the specification.
3. Solving the CSP with a finite domain solver.

Boolean constraints play an essential role since they drastically reduce the search space before the search and enumeration processes start on the generated CSP. Moreover, in the case where the program is not conforming with its specification, Boolean constraints provide a powerful tool for finding wrong behaviors in different execution paths of the program. An essential observation is that in this approach we do not transform all assignments and numerical instructions into Boolean constraints[2]. The point is that it is much more convenient to transform these instructions in finite domain constraints and to solve them with a CSP solver. So the collaboration between the SAT solver and the CSP solver is the cornerstone of our approach. Indeed, since we first identify the feasible paths, the finite domain solver will work with both smaller constraint systems and reduced domains.

The prototype system we have developed takes as input a JAVA program and its specification written in JML [15]. Currently, we only consider JAVA unit code without function calls, without return inside loops, and without inheritance. Moreover, we assume that all numerical operations only concern integers.

[2] Contrary to the most popular Model Checking approaches based on SAT Solvers [4, 5].

The rest of this paper is organised as follows. Section 2 gives an overview of our approach whereas Section 3 recalls some basics on constraint programming techniques. Section 4 details the verification process we propose and introduces the translation process we use to generate the constraint systems. Section 5 describes the experimental results and discusses some critical issues.

Before going into the details, let us illustrate the capabilities of our approach on a well-known benchmark.

2 Motivating Example

We illustrate our approach on the well-known `tritype` program for classification of triangles [7]. We first describe the program, then we show very informally how the transformation process works, and finally we describe different experimentations.

2.1 The Problem

The `tritype` program is a basic benchmark in test case generation since it contains numerous non feasible paths. `tritype` takes three positive integers as inputs (the triangle sides) and returns 1 if the inputs correspond to any triangle, 2 if the inputs correspond to an isoscele triangle, 3 if the inputs correspond to an equilateral one, 4 if the inputs do not correspond to any triangle. Figure 1 gives the `tritype` program in JAVA with its specification in JML. Note that `\result` in JML corresponds to the value returned by the program.

2.2 The Verification Process

We first translate this program and the negation of its specification into a set of constraints, using the process detailed in Section 4.2.

Then, in order to delay the enumeration on integers, we introduce boolean variables for each decision on input variables, e.g., we introduce variable eq_{ij} for condition $i = j$, variable eq_{ik} for condition $i = k$, and so on. So the resolution process is decomposed into two parts:

1. Finding a set of paths that correspond to potential non-conformities;
2. Solving the CSP which corresponds to the identified set of paths.

For instance, if the boolean solver finds the solution $\{eq_{ij} = true, eq_{jk} = true$, $eq_{ik} = false\}$ we generate the CSP $\{i = j, j = k, i \neq k\}$; the domain of i, j, k being {0,...,65635}. If the CSP has a solution we have found a test case which corresponds to a non-conformity. If none of the generated CSP has a solution the verification is done.

The constraints generated for lines 4 to 7 in Fig. 1 are displayed in Fig. 2. $cond \rightarrow c$ denotes a guarded constraint: roughly speaking, constraint c has to be satisfied when condition $cond$ holds (see section 3.3 for the exact semantic). $r0$ and $r1$ are the two first renamings of variable r. nul_i (resp. nul_j, nul_k) is the boolean variable that captures the decision $i = 0$ (resp. $j = 0$, $k = 0$). eq_{ij} is the boolean

```
/*@ public normal_behavior
  @ 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;
@*/

1  public static int tritype(int i, int j, int k){
2   int trityp ;
3   // not a triangle
4   if ((i==0)||(j==0)||(k==0)) trityp = 4 ; //ERR: trityp = 3
5   else {
6     trityp = 0 ;
7     if (i==j) trityp = trityp + 1 ;
8     if (i==k) trityp = trityp + 2 ;
9     if (j==k) trityp = trityp + 3 ;
10    if (trityp==0){
11       // triangular inequality not verified
12       if ((i+j <= k)||(j+k <= i)||(i+k <= j)) trityp = 4 ;
13       else trityp = 1 ; // any triangle
14    }
15    else {
16      if (trityp > 3) trityp = 3 ; // equilateral
17      else
18        //i=j and triangular inequality verified
19        if ((trityp==1)&&(i+j>k)) trityp = 2 ;
20        else
21          //i=k and triangular inequality verified
22          if ((trityp==2)&&(i+k>j)) trityp = 2 ; //ERR: (trityp == 1)
23          else
24            //j=k and triangular inequality verified
25            if ((trityp==3)&&(j+k>i)) trityp = 2 ;
26            else trityp = 4 ; // not a triangle
27          }
28      }
29    return trityp;
30  }
```

Fig. 1. `tritype` program in java with a specification in JML

variable for decision $i == j$. The constraint $((nul_i = 1) \vee (nul_j = 1) \vee (nul_k = 1)) \rightarrow r0 = 4$ corresponds to the `if` part of instruction on line 4 in Fig. 1, the following constraint corresponds to the else part. The last two constraints correspond to the `if` instruction on line 7 in Fig. 1. The full constraint system for the `tritype` program can be found in http://www.essi.fr/ rueher/appendix-tacas06.pdf.

// SSA variables for multiple definitions of result in the program

r0 : {0,...,65635}, r1 : {0,...,65635},

// boolean variables

nul_i : {0,1}, nul_j : {0,1}, nul_k : {0,1}, eq_{ij} : {0,1}

//constraints of line 4 to 7 of the program

$((nul_i{=}1) \vee (nul_j{=}1) \vee (nul_k{=}1)) \quad \rightarrow \text{r0=4}$

$\neg\ ((nul_i{=}1) \vee (nul_j{=}1) \vee (nul_k{=}1)) \quad \rightarrow \text{r0=0}$

$\neg\ ((nul_i{=}1) \vee (nul_j{=}1) \vee (nul_k{=}1)) \wedge (eq_{ij}{=}1) \quad \rightarrow \text{r1=r0} + 1$

$\neg\ ((nul_i{=}1) \vee (nul_j{=}1) \vee (nul_k{=}1)) \wedge \neg\ (eq_{ij}{=}1) \quad \rightarrow \text{r1=r0}$

Fig. 2. Constraints generated for lines 4 to 7 of the `tritype` program

2.3 Experimentations

We have introduced two errors into the `tritype` program:

1. A wrong return value when one of the inputs is zero (line 4 of the java program);
2. A wrong test on the trityp variable (line 22 of the java program).

These two errors occur in two different execution paths of the program. Figure 3 displays the four first non-conformities we have found: we successively display the path (i.e the value of decision variables), then three solutions of the corresponding integer system, and finally the value returned by the specification and the program.

The two first non-conformities are due to the wrong test on variable *trityp*, line 22 of the program. The first one is generated when “i=k”, and so “trityp=2”. Since the test on line 22 is “trityp==1” instead of “trityp==2”, the execution goes through the else part on line 25, so the value of result equals 4 instead of 2.

The second non-conformity corresponds to the case where “i=j”. So “trityp=1” and due to the wrong test on line 22 result equals 2 instead of 4 since the triangular inequality is verified.

The other errors we have found are those where at least one of the input is zero. Since we have introduced the error “trytyp = 3” instead of “trytyp = 4” on line 4 of Fig. 1, the program returns 3 instead of 4 whenever an input is equal to zero. The overall process finds 15 non-conformities in less than 5 seconds CPU time.[3]

We did also run the verification process with a correct program. It required 2.36 seconds CPU time to perform the complete verification. Note that we explored only 92 solutions of the Boolean constraint system although there are 9 variables, and thus 2^9 combinations. This clearly shows that the constraint system is strong enough to prune the search space, and to avoid a costly enumeration of all paths.

[3] All experimentations have been performed with ILOG Solver (see http://www.ilog.com/products/solver) and run on a Intel(R) Pentium(R) 4 CPU 2.00GHz computer with 256 Mb memory.

```
Error 1
Path : !(i=j), i=k, !(j=k), !(i+j≤ k), !(j+k ≤ i), !(i+k ≤ j), !(i=0), !(j=0), !(k=0)
Input values : i:2, j:1, k:2 – i:2, j:3, k:2 – i:3, j:1, k:3
Specification : 2, program : 4
Error 2
Path :i=j, !(i=k), !(j=k), i+j ≤ k , !(j+k ≤ i), !(i+k ≤ j), !(i=0), !(j=0), !(k=0)
Input values : i:1, j:1, k:2 – i:1, j:1, k:3 – i:1, j:1, k:4
Specification : 4, program : 2,
Error 3
Path : !(i=j), !(i=k), !(j=k), !(i+j ≤ k) , !(j+k ≤ i), i+k ≤ j, !(i=0), !(j=0), k=0
Input values : i:1, j:2, k:0 – i:1, j:3, k:0 – i:1, j:4, k:0
Specification : 4, program : 3
Error 4
Path : !(i=j), !(i=k), !(j=k), !(i+j ≤ k), j+k ≤ i, !(i+k ≤ j), !(i=0), !(j=0), k=0
Input values: i:2, j:1, k:0 – i:3, j:1, k:0 – i:3, j:2, k:0
Specification : 4, program : 3
```

Fig. 3. Four first non-conformities for the `tritype` program with two errors

3 Constraint Programming

This section recalls some basic concept of constraint programming which are useful to understand this paper. More details can be found in [21, 18, 13].

3.1 Definition of a CSP

Constraint programming is a paradigm that is tailored to hard search problems. The main application areas are planning, scheduling, timetabling, routing, placement, investment, configuration, design and insurance. Constraint programming incorporates techniques from mathematics, artificial intelligence and operational research; it offers significant advantages in these areas since it supports fast program development, economic program maintenance, and efficient runtime performance.

Constraint programming solvers are based on a *branch and prune* algorithm that combines local consistencies and efficient search heuristics.

More precisely, a *Constraint Satisfaction Problem* (CSP) is defined as:

– a set of *variables* $X = \{x_1, ..., x_n\}$,
– a finite set D_i of possible values for each variable x_i, called *domain*,
– a set of *constraints* $C = \{c_1, ..., c_n\}$ restricting the values that the variables can simultaneously take; X_j denotes the set of variables that occur in constraint c_j.

Note that the domains are a convenient way to express some specific constraints.

A *solution of a CSP* is an assignment of a value from its domain to every variable, in such a way that all constraints are satisfied.

3.2 Solving a CSP

To solve a CSP pruning techniques -that reduce the domain of the variables- are combined with search and enumeration heuristics. We only detail here local consistencies techniques.

Local consistencies are a key issue in finite domains where arc–consistency [19, 16] is very popular. A constraint c_j is arc-consistent if for any variable x_i in X_j, each value in D_i has a support in the domains of all other variables of X_j. In other words, a constraint c is arc–consistent for variable x, if values exist in the domains of all other variables such that constraint c holds when x is assigned to any value of its domain. The essential observation is that local consistency filtering algorithms try to reduce the size of the domain of some variable by considering only one constraint.

The following example shows in a very informal way how arc–consistency works. Consider the constraint system $C_1 = \{c_1 : x_1 + x_2 > 2, c_2 : x_1^2 + x_2^2 \leq 4, D_1 = \{0, 1, 2\}, D_2 = \{0, 1, 2\}\}$.

Constraint c_1 cannot be satisfied when either x_1 or x_2 are equal to 0. So arc–consistency will remove value 0 from domain D_1 and domain D_2. Now, constraint c_2 can no longer be satisfied when x_1 or x_2 are equal to 2, and thus value 2 will be removed from both domains. However, since the domain of one of the variables of constraint c_1 has been modified, we have to reconsider this constraint. Now, c_1 can no more be satisfied, the value 1 is removed from its domain which become empty; thus arc-consistency has detected the inconsistency of the whole constraint system.

Constraint system C_2 (see below) shows a case where the constraint system is arc-consistent but no solution satisfying all constraints exists.

$$C_2 = \{c_1 : x_1 \neq x_2, c_2 : x_2 \neq x_3, c_1 : x_3 \neq x_2\}, D_1 = D_2 = D_3 = \{0, 1\}.$$

3.3 Guarded Constraints

In this paper we also use guarded constraints. Guarded constraints are conditional constraints whose evaluation depends upon other constraints. $C_0 \rightarrow C_1$ denotes a guarded constraint where C_0 and C_1 are conjunctions of basic constraints. Relation $C_0 \rightarrow C_1$ states that constraints C_1 have to be added to the current constraint store when the solver can prove that constraints C_0 hold. More precisely, let C_0 be a boolean expression and C_1 a set of constraints, the guarded constraint $C_0 \rightarrow C_1$ behaves as follows:

- When the solver can prove that C_0 is true, then constraints C_1 are added to the store of constraints;
- When the solver can prove that C_0 is false, then the guarded constraint is just discarded;
- When the solver can neither prove that C_0 is true, nor prove that C_0 is false, that is when not enough variables of C_0 are instantiated, then the guarded constraint is suspended.

The solver tries to prove that the guard C_0 of a suspended constraint holds whenever the domain of some variable occurring in C_0 has been reduced. Of course, some guarded constraints may never become active.

One major difficulty with guarded constraints is that nothing can be done before the solver can demonstrate that the condition is either *true* or $false$. Let us consider a very simple piece of code:

```
//@ ensures \result ≥ 0
public int absolute(int i, int j) {
   if (i<j) return j-i;
        else return i - j;
}
```

This code is translated into the following set of constraints:

$$\{i < j \rightarrow r = j - i, \neg(i < j) \rightarrow r = i - j, r < 0, D_i = D_j = D_r = \{0, ..., 65635\}\}$$

A standard CSP solver cannot achieve any pruning on this system since nothing is known about i and j. So a very costly enumeration process is started: the inconsistency is only detected when the domain of i and j are reduced to one value. The advantages of combining SAT solver and CSP are obvious here. After having introduced a boolean variable for modeling $i < j$, the SAT solver enumerates the two paths, that is to say the two CSP $\{r = j - i, i < j, r < 0\}$ and $\{r = i - j, i \geq j, r < 0\}$. When the constraints of the CSP are transformed in binary constraints, arc-consistency immediately detects the inconsistency.

4 Verification Process

In this section we describe the overall verification process and explain how we transform the program and its specification into a set of constraints.

4.1 Verification Steps

The different operations which are performed during the verification process are detailed in Fig. 4.

Note that in step 3, we introduce boolean variables only to model decisions about input variables. This is sufficient to delay the enumeration process induced by guarded constraints (see Section 3.3). On the other hand, assignments are modeled using integer variables. Thus, we lose less information than with a translation of any statement into a boolean variable.

4.2 Translating the Program into a Set of Constraints

We first transform the program into its SSA form: for each new definition of a program variable, we introduce a fresh variable. In order to manage control instructions, we use ϕ–functions for `if then else` statements and we unfold loops. We use guarded constraints to model conditional execution flow (see part 3.3).

1. Put the program into a simplified Single State Assignment (SSA) form [14] and translate the SSA program into a set of constraints.
2. Add the constraints corresponding to the negation of the property to be proved.
3. Introduce a boolean variable for each decision on an input variable; Let *BoolSystem* be the constraint system obtained after steps 1, 2, and 3.
4. Start a solving process on *BoolSystem* and for `each` solution of *BoolSystem*:
    a. Build a CSP *IntSystem* that corresponds to the boolean values found in the current solution of *BoolSystem*
    b. Start a solving on IntSystem and for `each` solution of IntSystem print the current values of boolean variables (path trace) and find some errors of *IntSystem* (wrong input values).
5. If *BoolSystem* has no solution or if for each boolean solution *IntSystem* has no solution, print "the program is conform with is specification".

Fig. 4. Verification process

Basic Statements. Each assignment $var \leftarrow value$ is translated as a constraint $var = value$. Each boolean condition is translated as the corresponding constraint. We denote $SSA(s)$ the constraint corresponding to the basic statement s where each new definition of a variable has been replaced by the current renaming of this variable.

The `If then` Statement. For the sake of clarity, we only focus on the assignment of a single variable. Trivially, the same process could be applied individually for each variable appearing in a block with many variable assignments. Let us consider the statement `S : if (cond)` {`var=val1;var=val2; ...; var=valq;`}. Assume that *var* has already been defined p times before this statement. S is translated into the following set of guarded constraints:

$$\begin{array}{lll}
\text{SSA(cond)} & \rightarrow & var_{p+1} = \text{SSA}(val_1) \\
\text{SSA(cond)} & \rightarrow & var_{p+2} = \text{SSA}(val_2) \\
\ldots & & \\
\text{SSA(cond)} & \rightarrow & var_{p+q} = \text{SSA(valq)} \\
\text{// else part} & & \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+1} = var_p \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+2} = var_p \\
\ldots & & \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+q} = var_p
\end{array}$$

The else part is useful to ensure that the q fresh variables will not remain uninstantiated in the corresponding CSP.

The `If then else` Statement. Let us consider the statement `S : if (cond)` {`var=val11;var=val12; ...;var=val1q;`} `else` {`var=val21;var=val22;... var=val2r;`}. Assume that *var* has already been defined p times before this statement and assume that $q < r$. Since *var* has not the same number of definitions in the `if` part and the `else` part, we need to introduce a guarded constraint to take the place of the ϕ function. So, S is translated into the following set of guarded constraints:

$$
\begin{array}{lll}
\text{// if part} & & \\
\text{SSA(cond)} & \rightarrow & var_{p+1} = \text{SSA}(val_{11}) \\
\text{SSA(cond)} & \rightarrow & var_{p+2} = \text{SSA}(val_{12}) \\
\ldots & & \\
\text{SSA(cond)} & \rightarrow & var_{p+q} = \text{SSA}(val_{1q}) \\
\text{// else part} & & \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+1} = \text{SSA}(val_{21}) \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+2} = \text{SSA}(val_{22}) \\
\ldots & & \\
\text{SSA}(\neg\text{cond}) & \rightarrow & var_{p+r} = \text{SSA}(val_{2q}) \\
\text{// } \phi \text{ function} & & \\
\text{SSA(cond)} & \rightarrow & var_{p+q+1} = var_{p+q} \\
\text{SSA(cond)} & \rightarrow & var_{p+q+2} = var_{p+q} \\
\ldots & & \\
\text{SSA(cond)} & \rightarrow & var_{p+r} = var_{p+q}
\end{array}
$$

Remark: If $q > r$ the same principle is applied and the guarded constraints of the ϕ function are guarded by $SSA(\neg\ cond)$. If q=r then no ϕ function is required.

Figure 5 gives the translation of an overlapped `if then else`.

```
1  if (i < j) x = 0;
      else {
2         if (i < 30) {
             x = x+1;
             x = x+y;
          }
          else {
3            if (j > 43) x=2;
             else x=3;
          }
      }
```

$(i<j) \rightarrow x1=0$

$(\neg(i<j) \wedge (i<30)) \rightarrow (x1=x0+1 \wedge x2=x1+y0)$

$(\neg(i<j) \wedge \neg(i<30) \wedge (j>43)) \rightarrow x1=2$

$(\neg(i<j) \wedge \neg(i<30) \wedge \neg(j>43)) \rightarrow x1=3$

// ϕ-function for #2 if

$(\neg(i<j) \wedge \neg(i<30)) \rightarrow x2=x1$

// ϕ-function for #1 if

$(i<j) \rightarrow x2=x1$

Fig. 5. Example of if then else translation

The `Loop` Statement. We first transform any loop into the equivalent while loop. Then we unfold the while loop using an overestimate of the number of loop steps. This overestimate may be the worst case complexity of the loop or could be given by the user. To describe all possible paths inside the loop, we guard the constraints of the loop with the entrance condition. Our process is close to the one described in [4] except that we use guarded constraints instead of boolean operators to combine condition and assignment. More precisely, let us consider the following while loop `L : while (cond) var = value;`. We assume that this loop is executed at most max time and that var was defined p times before this statement. Then the loop statement L is translated into the following set of guarded constraints:

$$
\begin{array}{ll}
cond_1 & \rightarrow var_{p+1} = val_{p+1} \\
\neg\ cond_1 & \rightarrow var_{p+1} = var_p \\
cond_1 \wedge\ cond_2 & \rightarrow var_{p+2} = val_{p+2} \\
\neg\ (cond_1 \wedge\ cond_2) & \rightarrow var_{p+2} = var_{p+1} \\
..... & \\
cond_1 \wedge\ cond_2 \wedge\ ... \wedge\ cond_{max} & \rightarrow var_{p+max} = val_{p+max} \\
\neg\ (cond_1 \wedge\ cond_2 \wedge\ ... \wedge\ cond_{max}) & \rightarrow var_{p+max} = var_{p+max-1}
\end{array}
$$

where val_i denotes the ith SSA form of `value`.

With this system of guarded constraints, if the loop condition has never been true, then $var_{p+max} = var_p$, if it has been true only once then $var_{p+max} = var_{p+1} = val_{p+1}$, if it has been true max times then $var_{p+max} = val_{max}$.

Remarks

- The number of unfoldings may not be sufficient to detect all non-conformities especially when an error in the program entails more iterations than specified by the theoretical bound.

- When the bound of a for loop is well-known and when the index variable is not modified inside the loop block, it is more efficient to generate n constraint systems, one for each value of the decision variable. This is due to the fact that the guarded constraints are expensive to manage, even when the conditions are instantiated very early.

5 Experimental Results and Discussion

In this section we analyse the experimentations we have performed on three non-trivial academic examples.

5.1 The `tritype` Program

The first example we consider is the `tritype` program. As we mentioned in the introduction, we can find some errors in the program as well as prove the correctness of the program. Introducing boolean variables only for decisions on input variables gave very good results in this case. Indeed, `tritype` is a typical example of pure decisional program, so the proof mainly consists in showing that the same decision in program and specification gives the same code condition return value.

5.2 The `merge` Example

This `merge` program referenced in [11] computes five outputs from five inputs[4]. A partial order is given on the inputs and the property to be proved is that the

[4] The `merge` program and its JML specification can be found in http://www.essi.fr/~rueher/appendix-tacas06.pdf

outputs are sorted in decreasing order. In this program, a contrario to tritype program, the link between the specification and the program goes through the operational part. So we need to introduce boolean variables not only to model decisions but also to model the assignments. We have introduced the same error as in [11]. We found four error paths including the one shown in [11]. For each error path we search five different integer values for the inputs. The overall process took 159.71 seconds CPU time whereas the proof of the correctness required 310.67 seconds CPU time (no CPU time is given in [11]).

5.3 The `bsearch` Program

The `bsearch` program[5] takes as input an array of integers t sorted by increasing order, an integer value val to search in the array, and returns the index of the value if it is found or -1 otherwise. The worst case complexity of this program is $O(log_2(n))$ where n is the size of the array.

To perform the verification, we introduce boolean variables for condition tests on input variables (i.e $t[i] = val$, $t[i] < val$, $t[i] < t[i+1]$). Since the worst case complexity of bsearch is $O(log_2(n))$ we unfold the program loop $\lceil (log_2(n)) \rceil$ times. We successively introduce two errors. The first one is to return the value $middle + 1$ when the $t[middle] = val$. This error was detected by the CSP solver. The errors found by the solver correspond to all the possible paths through the loop when it stops with $t[middle] = val$. The second error consists in assigning the right bound with $middle$ instead of $middle + 1$ when $t[middle] < val$. With this second error the program will not terminate in some cases, for example when searching a value which is bigger than all the values in the array. This error was also detected.

The correctness proof was also performed. The required time exponentially increases according to the length of the array. The solver runs out of memory for arrays of size > 8 and values in $[0,2^{16}]$.

6 Discussion and Related Work

The new framework we have introduced in this paper has of course some limitations, e.g., there is no way to prove temporal properties, it works well with JAVA program but it would be difficult to handle C programs with pointers. Even for JAVA programs there are some restrictions: inheritance and functions calls are currently not handled[6].

A critical issue concerns the detection of inconsistencies in the CSP generated for each Boolean constraint system. Indeed, the constraints in some CSP may be too weak to achieve any pruning of the solution space, even when this CSP has no solution. In this case, a very costly search process is required to demonstrate

[5] The `bsearch` program and its JML specification can be found in http://www.essi.fr/~rueher/appendix-tacas06.pdf

[6] As long as we only consider finite structures, it should be possible to incorporate these features into our framework without major difficulties.

that the CSP is inconsistent. To overcome - at least partly - this problem some dedicated solvers could be used. For instance, when the finite domain constraints are linear, linear programming solvers could be used to reduce the domains. Formal simplifications of the constraint system could also be useful in some cases.

Of course, this problem is highly dependent from the modelling of the program and its specification. In other words, the kind of constraints that are generated will have strong influence on the performances of the solver [7].

Ganziger et al [8] have introduced a general DPLL(X) engine, where parameter X can be instantiated with a specialized solver $Solver_T$. That's to say DPLL(X) is a general engine for propositional solving. The authors illustrate their approach on their solver for EUF(logic with equality with uninterpreted functions). The goal of the approach introduced in this paper is not to integrate a CSP solver in a general DPLL engine. In our framework the essential role of the SAT solver is to boost the CSP solver by reducing the search space.

Armando et al [1] have recently proposed to use SMT solvers instead of SAT solvers for bounded model checking of software. We have compared our solver with their SMT-CBMC solver, which use CVC Lite for the theory of bit vectors. We have performed experimentations on the two sorting benchmarks contained in their last paper [1]. SMT-CBMC requires more than 600 seconds to analyse a bubble sort program with an array of size 26 whereas our solver analyses the same program with an array of size 100 in less than one second. Similarly, SMT-CBMC requires more about 200 seconds to analyse a selection sort program with an array of size 29 whereas our solver analyses the same program with an array of size 100 in less than 3 seconds.

We have also started an evaluation of our framework on standard SMT benchmarks (http://www.csl.sri.com/users/demoura/smt-comp/2005/). First results are promising; for instance we did prove the unsatisfiability of DTP_k2_n35_c210-s7.smt and DTP_k2_n35_c245_s10.smt in less than one second.

7 Conclusion

In this paper we have performed a first exploration of the capabilities of constraint techniques for verifying the conformity of a program with its specification.

First experimentations show that these techniques can be very efficient on some non trivial problems. Further work concerns the inclusion of dedicated solvers or simplifiers in our framework as well as a deeper study of the modelling issue.

Acknowledgements. Many thanks to Laurent ARDITI and Claude MICHEL for numerous and enriching discussions on this work.

[7] This is a well known problem in constraint programming: the performances of a solver may be very different on two constraint systems that are logically equivalent.

References

1. Armando, A., Mantovani, J., and Platania, L.: Bounded Model Checking of C Programs using a SMT solver instead of a SAT solver Technical Report, AI-Lab, DIST, University of Genova, December 19, 2005, 16 pages.
2. Ball T., Rajamani S. K., : Boolean Programs : A Model and Process For Software Analysis. Technical Report MSR TR 200-14, 2000
3. Bouquet, F., Dadeau, F., Legeard, B. and Utting, M: JML-Testing-Tools: a Symbolic Animator for JML Specifications using CLP. Procs of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05). Springer-Verlag. LNCS 3440, pp. 551–556, 2005.
4. Clarke E., Kroenig D., Lerda F. : A Tool for Checking ANSI-C programs. TACAS 2004, LNCS 2988, pp. 168-176, 2004
5. Clarke E., Kroenig 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
6. Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, Karen Yorav: SATABS: SAT-Based Predicate Abstraction for ANSI-C. Procs of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05). Springer-Verlag. LNCS 3440, pp. 570–574, 2005.
7. Demillo R. A., Offut A.J. : Experimental Results from an Automatic Test Case Generator. ACM Transactions on Software Engineering Methodology. vol. **2**, number 2, 1993, pp. 109-175
8. Ganzinger,H., Hagen,G., Nieuwenhuis, R.,Oliveras, A., and C. Tinelli: DPLL(T): Fast Decision Procedures. Proc. of CAV 2004. Springer-Verlag. LNCS 3114, pp. 175-188, 2004.
9. Gotlieb A., Botella B. and Rueher M : Automatic Test Data Generation using Constraint Solving Techniques. Proc. ISSTA 98, ACM SIGSOFT, vol. 2, pp. 53-62, 1998.
10. Gotlieb A., Botella B. and Rueher M : A CLP Framework for Computing Structural Test Data Proc of Computational Logic (CL2000), pp. 399-413, 2000.
11. Keller C. W., Saha D., Basu S., Smolka S.A. : FocusCheck : A tool for Model Checking and Debugging Sequantial C Programs. TACAS 2005, LNCS 3440, pp. 563-569, 2005
12. Kon O. and Castanet R. : Test generation for interworking systems. Computer Communications,vol. 23, pp. 642–652, 2000.
13. Krzystof R. Apt : Principles of Constraint Programming Cambridge University Press, 2003.
14. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
15. Leavens Gary T. and Cheon Yoonsik : Design by Contract with JML. www.jmlspecs.org, August 2005.
16. A. Mackworth : Consistency in networks of relations. *Journal of Artificial Intelligence*, pages 8(1):99–118, 1977.
17. Malay K. Ganai, Aarti Gupta, Pranav Ashar: DiVer: SAT-Based Model Checking Platform for Verifying Large Scale Systems. Procs of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05). Springer-Verlag. LNCS 3440, pp. 575–580, 2005.

18. Michela Milano (editor): Constraint and integer programming Kluwer Academic Publisher, 2004.
19. U. Montanari : Networks of constraints : Fundamental properties and applications to image processing. *Information science*, 7:95–132, 1974.
20. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik,M: Chaff: Engineering an Efficient SAT Solver. Proc of DAC, pp. 530–535, 2001
21. Rina Dechter: Constraint Processing. Morgan Kaufmann publisher, 2003
22. Sy N.T. and Deville Y.: Automatic test data generation for programs with integer and float variables. Proc of. 16th IEEE International Conference on Automated Software Engineering(ASE01), 2001.
23. Sy N.T. and Deville Y.: Consistency Techniques for interprocedural Test Data Generation. Proc. of the Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE03), Helsinki, Finland, 2003.