Exploring Different Constraint-Based Modelings for Program 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}@polytech.unice.fr

Abstract. Recently, constraint-programming techniques have been used to generate test data and to verify the conformity of a program with its specification. Constraint generated for these tasks may involve integer ranging on all machine-integers, thus, the constraint-based modeling of the program and its specification is a critical issue. In this paper we investigate different models. We show that a straightforward translation of a program and its specification in a system of guarded constraints is ineffective. We outline the key role of Boolean abstractions and explore different search strategies on standard benchmarks.

1 Introduction

Constraint programming techniques have been used to generate test data (e.g., [6,13]) and to develop efficient model checking tools (e.g. [10,4]). SAT based model checking platforms have been able to scale and perform well due to many advances in SAT solvers [11]. Recently, constraint-programming techniques have also been used to verify the conformity of a program with its specification [3,8].

To establish the conformity between a program and its specification we have to 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.

Experimentations reported in [8] demonstrate that constraint techniques can be used to handle non-trivial academic examples. However, we are far from the state where this techniques can be used automatically on real applications. Modeling is a critical issue, even on quite small programs. That's why we investigate different models in this paper.

The framework we explore in this paper can be considered as a very specific instance of SMT solvers[1]

[1] For short, a Satisfiability Modulo Theories (SMT) problem consists in deciding the satisfiability of ground first-order formulas with respect to background theories such as the theory of equality, of the integer or real numbers, of arrays, and so on [5,12,1].

The different models and search strategies we did experiment with, showed that a straightforward translation of a program and its specification in a system of guarded constraints is ineffective[2]. These experimentations also clearly outlined the key role of appropriate Boolean abstractions.

The verification of the conformity between a program and its specification is highly dependant on the programming language and the specification language. In this paper, we restrict ourseleves to *Java programs* and *JML specifications* (for "Java Modeling Language" see www.cs.iastate.edu/~leavens/JML).

To illustrate the advantages and limits of the models and search strategies we introduce, we will use the following examples:

– `S1` (see figure 1) : This is a very simple example of a Java program and its JML specification : it returns 1 if $i < j$ and 10 otherwise. `S1` will help us to understand how to introduce boolean abstractions. For each model, we will give the complete constraint system for `S1`.
– `S2` : This is also a very simple example derived from `S1`. The only difference with `S1` is that it returns the result of a calculus on input variables instead of constant values, i.e., it returns $i + j$ when S1 returns 1 (line 5 in figure 1) and $2 * i$ when S1 returns 10 (line 6 in figure 1). The main idea is to evaluate the impact of arithmetic during the resolution process.
– `Tritype` : This is a famous example in test case generation and program verification. This program takes three numbers that must be positive. These numbers represent the lengths of three sides of a triangle. It returns 4 if the input is not a triangle, 3 if it is an equilateral triangle, 2 if it is isosceles and 1 if it is a scalene triangle.
– `Tri-perimeter` : This program has the same control part than the *tritype* program. It returns -1 if the three inputs are not the lengths of triangle sides, else it returns the perimeter of the triangle. The specification returns $i+j+k$ while the program returns either $3 * i$, $2 * i + j$, $2 * i + k$, $2 * j + i$ or $i + j + k$.

The program and the specification of *tritype* and *tri-perimeter* can be found at www.polytech.unice.fr/~rueher/annex_tritype_triperimeter.pdf.

All the java programs of the examples in this paper conform to their JML specifications. This corresponds to the much difficult problem, since the search must be complete. Indeed, detection of non-conformity is much easier in practice (it stops when the first solution is found) even if the difficulty from a theoretical point of view is the same.

The rest of this paper is organised as follows. Section 2 recalls some basics on the translation process we have implemented to generate the constraint systems. Section 3 details the different models we propose whereas section 4 introduces different solving strategies. Section 5 describes the experimental results and section 6 discusses some critical issues.

[2] A full translation in Boolean constraints is also ineffective as soon as numeric expression occurs in the program or in the specification. Indeed, in this case we need to translate each integer operation into bit-vector calculus. Thus, even SMT solvers like SMT-CBMC [2] which use specialized solver for bit-vector calculus fail to solve some trivial problems.

```
/* @ public normal_behavior
   @ ensures ((i<j)=>\result=1) && ((i>=j)=>\result=10)
 */
int simple(int i, int j) {
  int result;
  int k=0;
  if (i <= j) k=k+1;
  if (k==1 && i!=j) result=1;
     else result =10;
  return result;
}
```

Fig. 1. S1 example : JML specification and Java program

2 Translation of a Program and its Specification into Constraints

This section recalls basic techniques for translating a Java program and its JML specification into a set of constraints.

For the sake of simplicity, we only consider here a very restricted form of Java and JML programs. For the JML specification, we restrict ourselves to normal behaviour, i.e., we do not consider exceptions such as overflows. We also assume that the JML specification contains an \`ensures` statement, a logical expression defining the post-condition. It may also contain a \`requires` statement defining the pre-condition. Likewise, we only consider Java program where all variables are integers and we assume that functions have only one return statement. Finally, we do not detail here the process for handling loops. Interested readers can find details on the way we handle loops and several JML statements such as \`forall` statement in [8].

2.1 Translation Process

We only recall here the basics which are required to understand this paper. More details on SSA form can be found in [9].

Translating the Program into a Set of Constraints. We first transform the program into its SSA "Single State Assignment" form [9]: 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.

Basic statements. Each assignment $var = value$ is translated as a constraint $var_i = value$ where i is the current number of definition of variable var. For example, the following piece of code $x = x + 1; y = x * y; x = x + y;$ is translated as the set of constraints : $\{x_1 = x_0 + 1, y_1 = x_1 * y_0, x_2 = x_1 * y_1\}$.

Conditional execution flow. Conditional execution flows are translated into *guarded constraints.* Guarded constraints are conditional constraints whose evaluation depends upon other constraints. $C_0 \to C_1$ 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.

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:
$\{i < j \rightarrow r = j - i, !(i < j) \rightarrow r = i - j, r < 0\}$

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.

That's why we introduce Boolean variables and handled in a better way guarded constraints (see part 3.2).

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=val}`. Assume that *var* has already been defined p times before this statement. S is translated into the following set of guarded constraints where $SSA(s)$ denotes the constraint corresponding to the SSA form of the basic statement s.

if part :	SSA(cond) $\rightarrow$	$var_{p+1} = \text{SSA}(val)$
else part :	SSA(!cond) $\rightarrow$	$var_{p+1} = var_p$

The else part just ensures that the var_{p+1} fresh variable will not remain uninstantiated in the corresponding CSP.

The `If then else` statement. Let us consider the statement `S : if (cond)` `{v=`x_1`;v=`x_2`; ...;v=`x_q`;} else {v=`y_1`;v=`y_2`;...; v=`y_r`;}`. Assume that v has already been defined p times before this statement and assume that $q < r$. Since v 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 :

// if part
SSA(cond) $\rightarrow$ $(v_{p+1}$=SSA$(x_1))$&$(v_{p+2}$=SSA$(x_2))$&...&$(v_{p+q}$=SSA$(x_q))$
// else part
SSA(!cond) $\rightarrow$ $(v_{p+1}$=SSA$(y_1))$&$(v_{p+2}$=SSA$(y_2))$&...&$(v_{p+r}$=SSA$(y_r))$
// ϕ function
SSA(cond) $\rightarrow$ $(v_{p+q+1}=v_{p+q})$&$(v_{p+q+2}=v_{p+q})$&...&$(v_{p+r}=v_{p+q})$

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 2 gives an example of translation of an overlapped `if then else`.

```
1     if  (i < j) x = 0;        (i<j) --> x1=0
      else {
2       if (i < 30)             (!(i<j)&(i<30))-->(x1=x0+1&x2=x1+y0)
           { x = x+1;
             x = x+y;}
        else {
3        if (j > 43) x=2;       (!(i<j)&!(i<30)&(j>43))--> x1=2
            else x=3;           (!(i<j)& !(i<30)& !(j>43))--> x1=3
        }                       // phi-function for #2 if
      }                         (!(i<j)&!(i<30)) --> x2=x1
                                // phi-function for #1 if
                                (i<j) --> x2=x1
```

Fig. 2. example of if then else translation

Translating the Specification. The JML specification we handle is decomposed into two parts : the `\requires` statement and the `\ensures` statement. The `\requires` statement is a logical expression on input variables and the `\ensures` statement is a logical expression both on input variables and `\result`, which denotes in JML the value returned by the method. We translate the JML specification in the following way :

– each logical expression is translated into the corresponding constraint,
– the `\result` JML variable is associated to a new variable of the CSP named *result* which establishes the link between the program and the specification,
– we add the constraints issued from the `\requires` statement,
– we add the negation of constraints issued from the `\ensures` statement.

2.2 Characteristics of a CSP for Software Validation

We define a CSP for software validation as a tuple formed with a set of *integer variables*, a set of *boolean variables* (possibly empty) used to improve performance of guarded constraints propagation, a set of *guarded constraints* issued from the program and the specification and an *abstraction table* which gives the correspondence between the boolean abstract variables and the integer expressions. This is detailed in figure 3.

1. *Variables*
   - INT_VAR : set of finite variables with domain [min,max]
   - BOOL_VAR : set of boolean variables with domain [0,1]
   - result $\in$ INT_VAR : the variable which makes the link between the program (Java return statement) and the specification (`\result` JML statement)
2. *Constraints*
   - PROG_CONST : set of guarded constraints from the program
   - REQUIRE_CONST : set of guarded constraints from the JML `\requires` statement
   - ENSURE_CONST : set of guarded constraints from the negation of the JML `\ensures` statement
3. *Abstraction table*
   - SEMANTICS(b_i) provides the finite domain constraint that is modelled by the boolean abstract variable b_i.

Fig. 3. CSP for software validation

3 Modeling Issues

We present here different models –from the less abstract one to the most abstract one– that we studied during our experiments. All the models are illustrated on example *S1* of figure 1. To help the reading, in the successive figures for example *S1* , we start by '*' the lines which have changed from one model to the next one. When solving the CSPs, all the models are also evaluated on the two more significant benchmarks *tritype* and *tri-perimeter*.

3.1 `INT_CSP`: A Model Without Boolean Abstraction

In this model, we do not introduce any boolean variable : the guarded constraints are couples (g,c) where g and c are expressions on integer variables. Figure 4 illustrates this model on example *S1*.

3.2 `HYBRID_CSP_1`: Using Boolean Abstraction for the Program Guards

In this model, we introduce a boolean variable for each guard involved in the constraints of the program. The advantage is that the enumeration process begins with boolean variables and so guards are evaluated first. In this way, a

```
1. Variables
   INT_VAR = {i,j,r_0,r_1,k_0,k_1} in [min,max]
   BOOL_VAR = empty
   result in [min,max]
2. Constraints
   PROG_CONST :
     1   r_0=0
     2   k_0=0
     3   i<=j --> k_1=k_0+1
     4   !(i<=j) --> k_1=k_0
     5   k_1=1 & i!=j --> r_1=1
     6   !(k_1=1 & i!=j) --> r_1=10
     7   result=r_1
   REQUIRE_CONST : empty
   ENSURE_CONST :
     8   !(((i<j) --> result=1) & ((i>=j) --> result=10))
3. Abstraction table : empty
```

Fig. 4. Example S1: INT_CSP

constraint c can be posted even if the integer variables involved in its guard are not instantiated. The introduction of boolean entails a *non standard* processing of guarded constraints. For instance, consider the guarded constraint $i < j \rightarrow k_1 = k_0 + 1$ and assume that the boolean variable b_0 is associated to the guard $i < j$. When b_0 is set to true, both constraints $i < j$ and $k_1 = k_0 + 1$ are added to the constraint store.

Figure 5 shows this model on example *S1*. With respect to figure 4 we have introduced two boolean variables, g_0 for the guard of constraints 3 and 4 and g_1 for the guard of constraints 5 and 6.

3.3 `HYBRID_CSP_2`: Using Boolean Abstraction for Expressions Appearing in Several Guards

One drawback of the previous model is that it looses too much semantics; for instance, it doesn't take into account that the same logical expressions may be involved in distinct guards. That's why we introduce in the model `HYBRID_CSP_2` a boolean variable for each sub-expression which appears several times in the guards of the program constraints or in the specification constraints.

In the previous modeling of *example S1* (see figure 5), the sub-expression $i < j$ appears both in guard g_0 and in the specification constraints. So we introduced a new boolean variable to abstract this expression (see figure 6).

3.4 `HYBRID_CSP_3`: Adding Boolean Abstraction for Expressions Involving the Variable *result*

In order to have a better link between the program and the specification, we extend the HYBRID_CSP_2 model by introducing a boolean abstraction for each

```
1. Variables
   INT_VAR = {i,j,r_0,r_1,k_0,k_1} in [min,max]
*  BOOL_VAR = {g_0,g_1} in [0,1]
   result in [min,max]
2. Constraints
   PROG_CONST :
      1   r_0=0
      2   k_0=0
*     3   g_0 --> k_1=k_0+1
*     4   !g_0 --> k_1=k_0
*     5   g_1 --> r_1=1
*     6   !g_1 --> r_1=10
      7   result=r_1
    REQUIRE_CONST : empty
    ENSURE_CONST :
      8   !(((i<j) --> result=1) & ((i>=j) --> result=10))
3. Abstraction table
*  SEMANTICS(g_0) = i<=j
*  SEMANTICS(g_1) = k_1=1 & i!=j
```

Fig. 5. Example S1 : HYBRID_CSP_1

expression on the *result* variable. This variable is part both of the program and the specification and so it can be helpful to cut some branches during the resolution process. Since we can only assign variable *result* once, we add a constraint which states that only one abstract variable can be true at the same time.

For example *S1*, we introduced two boolean variables in the model of figure 6 : one for the expression *result=1* and the other for the expression *result=10*; the link between the possible values of variable *result* is done by constraint 7 in figure 7.

3.5 BOOL_CSP: Boolean Model

In this model, we introduce a boolean variable for each expression in the program and the specification. This is the model which is used by SAT solvers when the expressions do not contain any arithmetic expression. If they do, basic arithmetic operations must also be modelled with boolean constraints.

4 Solving the CSP

We have explored various strategies for solving hybrid CSP with boolean and integer variables. These strategies are closely related to the models we presented in the previous section. We operated with JSolver4Verif[7]: it is a Java version of Ilog solver[3] with specific propagation rules based on congruence techniques. However, on all the examples contained in this paper the performances of JSolver are very similar to the one of Solver.

[3] See http://www.ilog.com/products/cp/.

```
1. Variables
    INT_VAR = {i,j,r_0,r_1,k_0,k_1} in [min,max]
*   BOOL_VAR = {g_0,g_1,g_2} in [0,1]
    result in [min,max]
2. Constraints
   PROG_CONST
      1  r_0=0
      2  k_0=0
*     3  (g_0||g_1) --> k_1=k_0+1
*     4  !(g_0||g_1) --> k_1=k_0
*     5  g_2 --> r_1=1
*     6  !g_2 --> r_1=10
      7  result=r_1
   REQUIRE_CONST : empty
   ENSURE_CONST
      8  !((g_0 --> result=1) & (!g_0 --> result=10)))
3. Abstraction table
*  SEMANTICS(g_0) = i<j
*  SEMANTICS(g_1) = i=j
*  SEMANTICS(g_2) = k_1=1 & !g_1
```

Fig. 6. S1 example : HYBRID_CSP_2

4.1 Solving an Integer CSP

In order to solve INT_CSP we only have to perform a search on the CSP. If a solution is found, then it is an error test case (a data that satisfies the constraints of the program and of the negation of its specification); otherwise the program is conform to its specification.

4.2 Solving a Hybrid CSP Using a CSP Solver

We show here two strategies for solving hybrid CSP for models HYBRID_CSP_1, HYBRID_CSP_2 and HYBRID_CSP_3:

- Strategy 1 : searching for all solutions.
  Roughly speaking, this strategy consists into searching a solution to the hybrid CSP and then to construct an integer CSP which has the semantics of the current solution of the hybrid CSP. This process is detailed in figure 9. The goal of this strategy is to take advantage of of the forward and backward propagation process on guarded constraints.
- Strategy 2 : enumerating on boolean variables only
  This is the same algorithm as *Strategy 1* except that in line 1 of figure 9 we start the search by enumerating on boolean variables only (i.e. on B_VAR variables only).

  This strategy is mandatory when we use models HYBRID_CSP_2 or HYBRID_CSP_3. In these models, some boolean variables depend from other boolean variables and do not appear directly in the constraints of the program and the specification. So they may remain uninstantiated during the search on the HYBRID_CSP.

```
1. Variables
    INT_VAR = {i,j,r_0,r_1,k_0,k_1\} in [min,max]
*   BOOL_VAR = {g_0,g_1,g_2,b_0,b_1} in [0,1]
    result in [min,max]
2. Constraints
    PROG_CONST :
       1  r_0=0
       2  k_0=0
       3  (g_0||g_1) --> k_1=k_0+1
       4  !(g_0||g_1) --> k_1=k_0
*      5  g_2 --> b_0
*      6  !g_2 --> b_1
*      7  b_0 + b_1 = 1
    REQUIRE_CONST : empty
    ENSURE_CONST :
*      8  !((g_0 --> b_0) & (!g_0 --> b_1))
3. Abstraction table
*  SEMANTICS[g_0] = i<j
*  SEMANTICS[g_1] = i=j
*  SEMANTICS[g_2] = k_1=1 & !g_1
*  SEMANTICS[b_0] = result=1
*  SEMANTICS[b_1] = result=10
```

Fig. 7. S1 example : HYBRID_CSP_3

5 Experimental Results

5.1 Experimental Results

In this section we evaluate the various models and strategies on the *S1* and *S2* programs and also validate our conclusions on the two more realistic examples *tritype* and *tri-perimeter*.

In each table, we consider signed integers and we give the time performance according to the number of bits they are coded with. Since our purpose is to compare different modeling issues, we use a time out limit of ten minuts (denoted '–' in the tables). Nevertheless, we give the solving time required for the largest size of integers we could handle in some cases (in italic). *bool* denotes the number of boolean solutions, that's to say the number of finite domain CSPs which are generated, and which have to be disproved to demonstrate the conformity between the program and its specification.

The constraint systems for these different programs have been generated automatically. Primary boolean expressions are stored on the fly in a hash map and each expression is replaced with a boolean variable if it is used more than once, depending on the model. All experiments were performed on a Processor Intel Core 2 Duo E6400 (2,13 GHz, 1G memory).

```
1. Variables
    INT_VAR = {i,j,r_0,r_1,k_0,k_1} in [min,max]
*   BOOL_VAR = {g_0,g_1,g_2,b_0,b_1,b_2,...,b_11} in [0,1]
    result in [min,max]
2. Constraints
    PROG_CONST :
*       1  b_0
*       2  b_1
*       3  (g_0||g_1) --> b_2
*       4  !(g_0||g_1) --> b_3
*       5  g_2 --> b_4
*       6  ! g_2 --> b_5
*       7  b_2 + b_3 = 1
*       8  b_4 + b_5 = 1
    REQUIRE_CONST : empty
    ENSURE_CONST :
*       9  !((g_0 --> b_4) & (!g_0 --> b_5))
3. Abstraction table
    SEMANTICS(g_0) = i<j              SEMANTICS(g_1) = i=j
    SEMANTICS(g_2) = k_1=1 & !g_1
*   SEMANTICS(b_0) = r_0=0            SEMANTICS(b_1) = k_0=0
*   SEMANTICS(b_2) = k_1=k_0+1        SEMANTICS(b_3) = k_1=k_0
*   SEMANTICS(b_4) = result=1         SEMANTICS(b_5) = result=10
```

Fig. 8. Example S1 : BOOL_CSP

Let HYBRID_CSP={I_VAR, B_VAR, CONST, SEMANTICS} where I_VAR is the set of integer variables including result*, B_VAR the set of boolean variables, CONST is the union of constraints from program and specification and SEMANTICS is the abstraction table*

```
boolean conform(HYBRID_CSP)
1 start a search on HYBRID_CSP
2 if HYBRID_CSP has no solution print program conform with its specification; return true
3 else
4     while HYBRID_CSP has a solution
5     - search next solution S of HYBRID_CSP
6     - build integer CSP S_INT :
          . for each variable I in I_VAR add a variable S_I in S_INT
            with initial domain equals to the domain of I in solution S
          . add the constraints of CONST where each variable I has been renamed as S_I
          . for each variable $B_i$ in B_VAR,
              if $B_i$ is true in solution S add the constraint SEMANTICS($b_i$)
              else   if $b_i$ is not an abstraction of an assignement of variable result,
                     then add the constraint (!SEMANTICS($b_i$))
7     - start a search on S_INT : if there is a solution it is an error test-case; return false
8 print program conform with its specification; return true.
```

Fig. 9. Strategy 1: solving an hybrid CSP

5.2 INT_CSP Model

As mentioned in subsection 3.2, the INT_CSP model cannot achieve any filtering as long as the integer variables involved in the guards are not instantiated. So, the search process is very slow: table 1 shows that even for the very simple examples, the INT_CSP model cannot be solved for integers coded on 32 bits.

Table 1. CSP_INT solving

# bit	*S1*	S2	tritype	tri-perimeter
8	0.577 s	0.766 s	66.582 s	406.27 s
10	5.422 s	9.255s	-	-
16	*21663.778 s (6 hours)*	-	-	-
32	-	-	-	-

5.3 Hybrid CSP Models

Table 2 provides the results for *S1* and *tritype* programs.

In these programs, the returned value is a constant so the `HYBRID_CSP_3` is not relevant. Indeed, in this case there is little interest to introduce a boolean abstraction for each expression on the *result* variable.

Searching all solutions (strategy 1) is rather inefficient with the `HYBRID_CSP_1` model. Strategy 2 (searching only boolean solutions) is clearly better both with `HYBRID_CSP_1` and `HYBRID_CSP_2`.

An essential observation is that strategy 1 searches for all the solutions, that is to say, even for solutions which differ on integer variables but are equals for boolean variables. As said before, we evaluated this strategy because the inverse propagation of guarded constraints may eliminate values for boolean variables on some problems. Indeed, for a guarded constraint (g, c), if c is proved false due to other constraints, then the negation of g is added to the constraint store; this information may cut some branches for other guarded constraints which share guard g.

The difference between the results for the various models highlights that introducing boolean variables is a key issue when these variables are shared by many constraints.

Table 3 compares the performances of models HYBRID_CSP_2, HYBRID_CSP_3 and BOOL_CSP on the *S2* and *tri-perimeter* examples using strategy 2.

First, let us note that the performances are weaker on these two examples. This is due to the arithmetic operations, which occur in these two examples. Indeed, *S2* (resp. *tri-perimeter*) differs from *S1* (resp. *tritype*) only on the operative part (calculus on inputs instead of constant value).

Another essential observation is that in model `BOOL_CSP`, we provide a constraint which states that the boolean variables which correspond to several assignements of a single variable cannot be true at the same time. This is a critical point: if we remove this constraint for *tri-perimeter* with integers coded on 8 bits there are 778240 boolean solutions and it takes 319.028s to solve the problem.

Table 2. HYBRID_CSP_1 and HYBRID_CSP_2 solving

S1 and tritype						
# bit	HYBRID_CSP_1 strategy 1		HYBRID_CSP_1 strategy 2		HYBRID_CSP_2 strategy 2	
	S1	*tritype*	*S1*	*tritype*	*S1*	*tritype*
8	57.116 s 131072 bool	-	0.194 s 4 bool	0.999 s 565 bool	0.182 s 4 bool	1.768 s 4520 bool
10	-	-	0.221 s 4 bool	4.157 s 565 bool	0.186 s 4 bool	1.926 s 4520 bool
16	-	-	0.568 s 4 bool	-	0.221 s 4 bool	8.522 s 4520 bool
32	-	-	-	-	*1520.82 s* 4 bool	-

Table 3. HYBRID_CSP_2, HYBRID_CSP_3 and BOOL_CSP solving

S2 and tri-perimeter						
# bit	HYBRID_CSP_2		HYBRID_CSP_3		BOOL_CSP	
	S2	*tri-perimeter*	*S2*	*tri-perimeter*	*S2*	*tri-perimeter*
8	0.477 s 8 bool	15.056 s 5056	0.185 s 4 bool	6.57 s 22464 bool	0.2 s 4 bool	3.42 s 6080 bool
10	2.946 s 8 bool	-	0.2 s 4 bool	10.489 s 22464 bool	0.286 s 4 bool	3.654 s 6080 bool
16	-	-	0.274 s 4 bool	-	0.292 s 4 bool	4.809 s 6080 bool
32	-	-	*2516.156 s* 4 bool	-	-	-

6 Discussion

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. However, such a verification remains a difficult task, even for small programs. Our experimentations [8] showed that constraint techniques can be very efficient on some non trivial problems. Performance of CSP techniques behave clearly better than state of art SMT solvers [8,2]

In this paper, we did investigate different CSP models on a few simple but non-trivial academic examples. As expected, a straightforward translation of a program and its specification in a system of guarded constraints is ineffective, even on very simple examples. Boolean abstraction is clearly a critical issue for efficiency. An appropriate Boolean abstraction is an essential support for the search process.

Of course, additional work is required before these techniques can be used on real applications. Further work to try is on the cooperation of CSP and SAT solvers as well as new filtering techniques.

When the set of Boolean constraints becomes larger, a collaboration between a SAT solver and CSP solver is probably more appropriate to handle such problems. We have performed some very preliminary experimentation with SAT4J(see www.sat4j.org) and Jsolver4Verif. A technical difficulty concerns the enumeration of all solutions by a SAT solvesr. Indeed, the most efficient SAT solver are not designated to enumerate all solutions. Moreover, the transfer to the SAT solver of failure information from CSP solver –which is a key issue– is far from being obvious.

Specific filtering techniques[4] may also drastically improve the refutation of the generated CSP over finite domains. Likewise, linear solvers or difference constraint solvers may be used to check the consistency of constraint defining the semantics of guards.

References

1. Aït-Kaci, H., Berstel, B., Junker, U., Leconte, M., Podelski, A.: Satisfiability Modulo Structures as Constraint Satisfaction: An Introduction. In: Procs. of JFLA, 8 pages (2007)
2. Armando, A., Mantovani, J., 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, 16 pages (December 19, 2005)
3. Bouquet, F., Dadeau, F., Legeard, B., Utting, M.: JML-Testing-Tools: a Symbolic Animator for JML Specifications using CLP. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 551–556. Springer, Heidelberg (2005)
4. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
5. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
6. Gotlieb, A., Botella, B., Rueher, M.: Automatic Test Data Generation using Constraint Solving Techniques. In: Proc. ISSTA 98, ACM SIGSOFT, vol. 2, pp. 53–62 (1998)
7. Leconte, M., Berstel, B.: Extending a CP Solver with Congruences as Domains for Program Verification. In: Procs. of CSTVA06, 1st Workshop on Constraints in Software Testing, Verification and Analysis, Nantes (2006)
8. Collavizza, H., Rueher, M.: Software Verification using Constraint Programming Techniques. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 182–196. Springer, Heidelberg (2006)
9. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, K.: Efficently Computing Static Single Assignment Form and the Control Dependence Graph. Transactions on Programming Languages and Systems 13(4), 451–490 (1991)

[4] See for instance [7] where congruence domains are introduced.

10. Ganai, M., Gupta, A., Ashar, P.: DiVer: SAT-Based Model Checking Platform for Verifying Large Scale Systems. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 575–580. Springer, Heidelberg (2005)
11. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. of DAC, pp. 530–535 (2001)
12. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT an SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM (to appear)
13. Sy, N.T., Deville, Y.: Automatic test data generation for programs with integer and float variables. In: Proc of. 16th IEEE International Conference on Automated Software Engineering(ASE01). IEEE Computer Society Press, Los Alamitos (2001)