Specification Analysis and Test Data Generation by Solving Boolean Combinations of Numeric Constraints*

Jian Zhang

Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing 100080, China
Email: zj@ox.ios.ac.cn

Abstract

In the analysis of state-based requirement specifications and in software test data generation, one often has to determine the satisfiability of Boolean combinations of numeric constraints. Theoretically this problem is undecidable, and practically many problem instances are very difficult. In this paper, an extension to a Boolean constraint solver is described. The new tool can deal with variables of other types (including the enumerated type, integers and reals). It combines Boolean logic reasoning with linear programming and bound propagation. Non-linear constraints are allowed, but the solving method is incomplete. Experimental results show that the tool can be quite useful in testing specifications as well as procedural code.

1 Introduction

During the software development process, many errors may occur. For example, incomplete requirements, design errors, bugs in source code, and so on. Various methods and tools can be used to detect these errors, such as visualization, static analysis and debugging. In this paper, we study automated techniques for the analysis of formal requirement specifications and for the generation of test data.

It is very important to discover flaws in the requirements. This may reduce the cost of testing significantly. For nontrivial software systems, it is desirable if the requirements can be formalized. There are several kinds of formal methods. Some of them are very expressive, while others can be easily automated.

A common way of specifying the requirements is to use some kind of state-transition rules. In such a specification, the system changes from one state to another when certain conditions are met or when some events happen. The conditions are essentially logic formulas. However, for the sake of readability, tables or diagrams are often used. For example, condition tables and event tables are employed in the Software Cost Reduction (SCR) requirement method [11]. In the Requirements State Machine Language (RSML) [8, 9], the conditions on transitions are represented by AND/OR tables. Such a table is equivalent to a formula in the disjunctive normal form. In general, a state-based specification should have the following properties [9, 11]:

- *Consistency* or *Disjointness*: For any state s, the conjunction (i.e., logical AND) of the conditions on every pair of transitions out of s is unsatisfiable.
- *Completeness* or *Coverage*: For any state s which is not the terminal state, the disjunction (i.e., logical OR) of the conditions on every transition out of s is a tautology.

Errors in the specification can be found if certain formulas are shown to be satisfiable. Suppose there is some state s with conditions $c_1, c_2, \ldots, c_m$. When c_i is satisfied, the system changes to the state w_i ($1 \leq i \leq m$). The specification is inconsistent if for example, (c_1 AND c_2) is satisfiable. The reason is that, the system can either move to w_1 or to w_2. This kind of nondeterminism is often considered harmful. On the other hand, if the formula NOT (c_1 OR c_2 OR ... OR c_m) is satisfiable, the specification is incomplete, because the next state of the system is not specified in all cases.

In specification-based testing (e.g., [15]), we also need to check the satisfiability of some constraints. The same is true for white-box testing. In path-oriented test data gen-

*Partially supported by the Natural Science Foundation of China (NSFC) under grant No. 69703014 and the National Key Basic Research Plan (NKBRSF) under grant No. G1998030600.

eration and feasible path analysis [5], it is necessary to determine the values of input variables such that the program is executed along a given path. The problem is essentially satisfying a set of formulas. This is discussed in some detail in [22]. But the automatic tools described there are not so appropriate for many testing benchmarks.

In the most general form, automatic test data generation and specification analysis are undecidable problems. No algorithm works in all cases. Difficulties arise when the program or the specification contains real variables and nonlinear expressions. Thus the problems are often simplified. A commonly used method is to represent various conditions by Boolean variables. Previously, Boolean methods and tools have been used by many researchers in the analysis of specifications and in test data generation. For example, when analyzing state-based requirement specifications, Heitmeyer *et al.* [11] use a tableaux-based decision procedure, while Heimdahl and Leveson [8, 9] use Binary Decision Diagrams (BDDs). As for software test data generation, the approach of Weyuker *et al* [20] is based on Boolean specifications, and all variables in the example of [5] are Boolean. The SpecTest tool developed in George Mason university only allows Boolean variables in the transition predicates.

Boolean formulas are relatively easy to analyze. However, we may obtain inaccurate results when replacing nontrivial conditions by Boolean variables. For example, false error reports often occur in the analysis of requirement specifications [8, 9]. In this paper, we study the satisfiability of formulas, where the variables can be Boolean or enumerated, and the Boolean variables may represent numeric constraints between real-valued variables. In the next section, we briefly describe an existing Boolean constraint solver and the transformation of enumerated types. Then in §3, we discuss how to combine the solver with a linear programming package. Examples are given to illustrate the ideas and the use of the new tool. Finally, our approach is compared with other related work, and directions for future research are suggested.

2 A Boolean Constraint Solver

Checking the satisfiability of Boolean formulas can be regarded as a special case of Constraint Satisfaction Problems (CSPs). In a CSP, we have a finite number of variables, each of which is associated with a domain (i.e., a set of values). In addition, there are some constraints defined on the variables (e.g., $x + y < 5$). For every variable, we need to find a value from its domain, such that when the variables are replaced by the corresponding values, all the constraints are satisfied. In many cases, the domains are finite and the problem is called a finite CSP.

Traditionally CSPs are solved by backtracking algorithms. The basic steps of such an algorithm are as follows:

(1) Choose a variable x which does not have a value yet. If every variable has a value, then a solution is found, and the algorithm terminates.

(2) Choose a value from the domain of x and assign it to x.

(3) Deduce new information from this assignment. If contradiction is found, we need to select another value for x or backtrack to a previous variable. When there is no previous variable, the algorithm terminates, without finding a solution.

Initially no variable has a value.

There are many ways of improving the performance of the above algorithm, e.g., choosing the variables in some particular order, deducing as many new assignments as possible after a value is assigned to a variable.

Based on the constraint solving paradigm, we implemented a Boolean expression satisfier [21] in C. It is called BOX later. In addition to the common Boolean operators such as NOT, OR, AND, the tool also allows the user to specify constraints of the form: LTEk(arg_1,arg_2,...,arg_n). It means that, among the n arguments, there are at most k which are true. (LTE stands for "less than or equal to".) For a simple example, given the following input:

```
{
   or(and(a1,a2,a3),and(b1,b2));
   LTE2(a1,a2,a3,b1,b2);
}
```

only one solution will be found, namely,

```
a1 = a2 = a3 = 0,   b1 = b2 = 1.
```

Here all the identifiers (except for keywords) are treated as Boolean variables. In the solution, 0 and 1 stand for FALSE and TRUE, respectively. Note that we use the prefix notation to express Boolean formulas. So `or(x,y,z)` is equivalent to $(x \vee y \vee z)$. The former is more convenient if each logical operator has many arguments, but the latter is more readable to most people. Also, the formulas are separated by semicolons. Every formula has to be satisfied.

For many applications, Boolean formulas are inadequate or inappropriate. So we extend BOX to deal with variables of other types, such as real variables and enumerated variables.

Enumerated Types

Heimdahl and Leveson's prototype tool may produce spurious error reports (i.e., false alarms), when there are non-Boolean variables. See Sec 5.1.1 of [9]. One common data type is the enumerated type, e.g.,

DaysW = { Mon, Tue, Wed, Thu, Fri, Sat, Sun }.

The values of such a type are all inclusive and mutually exclusive.

To eliminate false alarms caused by enumerated types, we may translate the variables and constraints into Boolean ones. The translation is straightforward. Given an enumerated variable $v : \{e_1, e_2, \ldots, e_n\}$, we introduce n new Boolean variables $\$ve_1$, $\$ve_2$, ..., $\$ve_n$. Then the condition "$v = e_i$" is replaced by $\$ve_i$, and the condition "$v$ **isOneOf** $\{e_1, e_2, \ldots, e_k\}$" is equivalent to $(\$ve_1 \vee \$ve_2 \vee \ldots \vee \$ve_k)$. Of course, for each enumeration type, one has to add two formulas indicating that the values are all inclusive and mutually exclusive:

$$\$ve_1 \vee \$ve_2 \vee \ldots \vee \$ve_n, \\ LTE1(\$ve_1, \$ve_2, \ldots, \$ve_n).$$

Example 1. Given the following input to our tool:

```
enum OwnAirStatus: Airborne, OnGround;
bool b1 = (OwnAirStatus == Airborne);
bool b2 = (OwnAirStatus == OnGround);
{ or(b1,b2); }
```

only two (rather than three) solutions will be found:

```
    b1 = 1, b2 = 0;    b1 = 0, b2 = 1.
```

We also tried several other examples, and the results are encouraging. As mentioned earlier, Heimdahl and Leveson's prototype tool may generate false alarms. An example is given in Figure 8 of [8]. Their tool reports that a problem exists when some condition is satisfied. The condition, if expressed as an AND/OR table, has 13 rows and 14 columns. On a SUN SPARCstation 5 with 16MB memory, it takes about 0.01 second for our tool to conclude that the condition is not satisfiable. We shall not describe the experiments in more detail.

3 Numerical Constraints

In most applications, real-valued variables are involved in the specification and the program. Thus in the analysis, we need to check the satisfiability of more complicated conditions. Such a condition is basically a Boolean formula, but each Boolean variable may correspond to a numeric constraint of the form $Exp_1\ rop\ Exp_2$. Here Exp_1 and Exp_2 are polynomials in the real domain (e.g., $2x - yz$), rop is a relational operator (e.g., $=, \leq, \neq$). This kind of numeric constraints will be called *primitive constraints*. The constraint corresponding to the Boolean variable b will be denoted by $pCons(b)$. A constraint is called a *simple constraint* if it has the form $x\ rop\ v$, where x is a real variable, v is a value, and rop is a relational operator.

When numeric constraints are involved, a solution to the Boolean formula is not necessarily a solution to the whole problem, and false alarms may occur in specification analysis. As a simple example, suppose the formula is $(b_1 \vee b_2)$, $pCons(b_1)$ is $x \geq 0$, and $pCons(b_2)$ is $2x - y^2 > 0$. Then the following Boolean solution is a false alarm: { b_1 is FALSE, b_2 is TRUE }, because when x is negative, $2x - y^2$ is also negative. A simple way to deal with the issue is to examine every solution manually, as described by some authors. But clearly, this is very tedious when the Boolean formula has many solutions. In this section, we describe some *automatic* analysis techniques and a tool for solving Boolean combinations of numeric constraints.

For simplicity, we assume that all the numeric variables are non-negative. We also assume that primitive constraints take the form of inequalities. Otherwise, suppose that there is a Boolean variable b_i, and $pCons(b_i)$ is the equation $Exp_1 = Exp_2$. Then we can introduce two new Boolean variables:

$$b_{i_1} : Exp_1 \leq Exp_2, \qquad b_{i_2} : Exp_1 \geq Exp_2$$

and replace b_i by $and(b_{i_1}, b_{i_2})$. Of course, we need to add the condition $or(b_{i_1}, b_{i_2})$ to the Boolean formula. Otherwise, we might get a solution in which b_{i_1} and b_{i_2} are both false. Obviously, it is impossible that $(Exp_1 > Exp_2)$ and $(Exp_1 < Exp_2)$ hold at the same time.

Bound Propagation

An important technique in numerical constraint solving is the so-called "bound propagation", i.e., deducing new bounds from the known bounds of some variables or expressions. A few examples of such reasoning are given as follows.

- If the lower bounds for $expr_1$ and $expr_2$ are lb_1 and lb_2, respectively, then we conclude that the lower bound for $(expr_1 + expr_2)$ is $(lb_1 + lb_2)$.
- If the lower bound for $expr_1$ is lb_1 and the upper bound for $expr_2$ is ub_2, then the lower bound for $(expr_1 - expr_2)$ is $(lb_1 - ub_2)$.
- If there is a constraint $expr_1 > expr_2$, and we deduced that the upper bound for $expr_1$ is ub_1 and the lower bound for $expr_2$ is lb_2 which is greater than ub_1, then we get a contradiction.

One can easily come up with other examples. They are also available from the literature (e.g., section 10.2 of [12]).

In real-world applications, there are lower and/or upper bounds for the values of most variables (e.g., the height of an airplane, the water pressure).

Linear Programming

Bound propagation is very simple, and it can be quite useful in certain cases. However, using this technique alone does not solve many problems. For instance, it cannot tell us that the equation $x = -x$ has exactly one solution if the range of x is $(-\infty, +\infty)$.

Since linear constraints appear more often in most programs, we decide to combine Boolean constraint satisfaction with linear programming (LP). An LP problem consists of a set of real-valued variables, a linear function (called the *objective function*) and a set of linear inequalities or equations (called *constraints*).

In our experiments, we use the LP package `lp_solve` [1] developed by Michel Berkelaar. Given the following input:

```
min: x + y;
x + 5y <= 5.3;
2x + y >= 8.2;
```

`lp_solve` will find the optimal value 4.1 for the objective function. Note that all variables are assumed to be non-negative.

Some LP problems may be *infeasible* or *unbounded.* The former means that the set of linear constraints is unsatisfiable, and the latter means that the objective function does not have a finite bound. The unbounded case is less interesting to us.

Checking Non-linear Constraints

Given a set S of primitive constraints, if all of the constraints are linear inequalities, then linear programming suffices to decide the satisfiability of S. Otherwise, we can combine bound propagation with LP to determine whether some non-linear constraints are unsatisfiable.

Suppose $S = C_L \cup C_N$, where C_L is the set of linear constraints, and C_N is the set of non-linear constraints. A constraint $nlc \in C_N$ can be tested in the following way. For each real-valued variable x_i in nlc, we solve the following two LP problems:

(1) minimize x_i, subject to C_L.

(2) maximize x_i, subject to C_L.

After obtaining the lower and upper bounds for the variables, we use bound propagation to check whether nlc holds. Of course, some bounds may be $-\infty$ or $+\infty$, and the method is incomplete. But in practice, it can be quite useful.

Example 2. Suppose we have three Boolean variables b_1, b_2 and b_3. The corresponding numeric constraints are as follows: $pCons(b_1) = (xy > 2)$, $pCons(b_2) = (2x \geq y + 3)$, $pCons(b_3) = (y < 1.2)$. One Boolean formula that has to be satisfied is $and(not(b_3), or(b_2, b_3))$. It is easy to see that the Boolean variables b_2 and b_3 should be true and false, respectively. So we get the lower bounds: $x \geq 2.1$, $y \geq 1.2$. Through bound propagation, we can conclude that b_1 should be true, which solves the whole problem immediately. On the other hand, if the definition of b_3 were $(x > 1.2)$, `lp_solve` would tell us that there is a contradiction.

When Is Numeric Reasoning Performed?

The primitive constraints can be used in one of the following ways:

(1) Postprocessing. Solve the Boolean part of the problem first. For each of the solutions, decide whether the corresponding set of numeric constraints is satisfiable.

(2) Preprocessing. Discover some logical relationships among the primitive constraints, and extend the Boolean formula accordingly. Then the Boolean constraint solver is used. Fewer solutions should be found.

(3) Modify the Boolean constraint solving algorithm so that it may use information in the numeric constraints.

Of course, they can be combined in a single framework. One benefit of the first two ways, as compared with the third one, is that we do not have to modify the source code of existing LP and satisfiability checking software. We shall describe them through an example.

Example 3. Suppose we have the formula $or(b_1, b_2)$. The Boolean variables b_1 and b_2 stand for $x > 2$ and $x^2 + y^2 \leq 4$, respectively. Postprocessing means that we first obtain 3 solutions of the Boolean formula: $b_1 = 1$, $b_2 = 1$; $b_1 = 0$, $b_2 = 1$; $b_1 = 1$, $b_2 = 0$. Then for each of them, we check whether the resulting inequalities hold. The first solution corresponds to: $x > 2$, $x^2+y^2 \leq 4$. They are inconsistent for real-valued variables x and y. Then we look at the second: $x \leq 2$, $x^2 + y^2 \leq 4$. No inconsistency is found. So this can be included in the report to the user. Similarly for the third solution.

In the case of preprocessing, we first notice that, for any $x > 2$, $x^2 + y^2 \leq 4$ does not hold. So we add the following to the Boolean formula: $not(and(b_1, b_2))$, and we get two solutions: $b_1 = 0$, $b_2 = 1$; $b_1 = 1$, $b_2 = 0$.

It is beneficial to adopt the postprocessing approach if the Boolean formula is unsatisfiable or has only a few solutions. Otherwise, it is too tedious to check each solution.

Generally speaking, it is not so effective to do preprocessing alone, because we cannot use the Boolean constraint to prune the search space. In fact, it corresponds to the case that the Boolean formula is TRUE. It may happen that, you spend a lot of time discovering some relationships between the numeric constraints, but the Boolean formula is easily determined to be unsatisfiable. However, some preprocessing methods can still be used, when we try to solve a set of problem instances which share the same set of primitive constraints. This occurs, for example, when we are generating a suite of test data.

Besides preprocessing and postprocessing, we may modify the Boolean constraint solver, so that it can use the numeric information during the search. The affected parts of the search algorithm include: heuristics for choosing the next variable to instantiate, and propagation methods which use the information in the primitive constraints to detect contradiction or to infer new values for other Boolean variables (as shown in Example 2).

BoNuS: The New Tool

In each of the above three ways, we have to solve the following type of problem. Given an assignment of truth values to (some) Boolean variables, decide if the resulting numeric constraints are consistent. For instance, if $pCons(a)$ is $x < 2$, $pCons(b)$ is $x^2 < 1.5$, and a is FALSE, b is TRUE, then there is a contradiction. To detect this, we use the techniques described earlier in this section.

We extended the Boolean constraint solver BOX, and the new tool is called BoNuS. It has several options. For example, the user can give a number m so that the tool stops after m solutions are found. There is another option for postprocessing. This may be specified when the Boolean formula does not have too many solutions.

We also modified the search procedure of BOX. To employ bounds for the real variables, we implemented the following search heuristic. When choosing a Boolean variable to assign a truth value, we give priority to those which represent simple constraints. When the values of these variables are fixed, we apply bound propagation to deduce the truth or falsity of other conditions, thus obtaining the truth values of the corresponding Boolean variables.

After simple constraints, the second highest priority is given to other linear constraints. For each of them, we compute a weight which depends on the variables in the constraint. The weight of a variable is larger if it occurs in more constraints. During the search, we favor constraints with larger weights.

Example 4. Suppose we have the following problem:

```
real x, y, z;
bool b1 = (10*x-y > 1);
bool b2 = (y-2*z < 5);
bool b3 = (x <= 0.5);
bool b4 = (y >= 3);
bool b5 = (z < 1);
{
  not( or(and(b1,b2),b3,b4,b5) );
}
```

It takes our tool a fraction of a second to find that the constraint is not satisfiable. In other words, the expression $((b_1 \wedge b_2) \vee b_3 \vee b_4 \vee b_5)$ is valid when the Boolean variables are defined as above. The theorem can be proved by PVS [16] in about 4 seconds. But PVS gets into difficulties if we change the definition of b_1 to be $(x - y > 1)$. In this case, our tool reports that there is one solution: b_2 is TRUE and all other Boolean variables are FALSE.

It should be noted that PVS is a general-purpose tool which can do many things, while our tool is mainly used to test the satisfiability of certain constraints.

Example 5. Consider the `middle` routine of [18]. (There are a few typos in the original version of the specification and the program, which are corrected here.) Given 3 integers `A`, `B` and `C`, the routine returns the middle numeric value. But if two of the input integers are equal, the other integer should be returned. Let us denote by `M` the desired return value. Then its relation with the input is as follows.

```
((B < A and A < C) -> M = A) and
((C < A and A < B) -> M = A) and
((A < B and B < C) -> M = B) and
((C < B and B < A) -> M = B) and
((A < C and C < B) -> M = C) and
((B < C and C < A) -> M = C) and
(B = C -> M = A) and
(A = C -> M = B) and
(A = B -> M = C)
```

An incorrect implementation is given below:

```
if ((A < B and B < C)
 or (C < B and B < A))
   return B;
else if ((A < C and C < B)
      or (B < C and C < A))
   return C;
else if ((B < A and A < C)
      or (C < A and A < B))
   return A;
else return A;
```

Let us denote by `N` the return value of the implemented routine. Then it can be equal to one of the 3 input integers, under different conditions. To check if the implementation violates the specification, we give the following input to BoNuS:

```
int  a, b, c, m, n;

bool ba = (b < a);
bool ac = (a < c);
bool ca = (c < a);
bool ab = (a < b);
bool bc = (b < c);
bool cb = (c < b);
bool a_b = (a == b);
bool b_c = (b == c);
bool c_a = (c == a);
bool m_a = (m == a);
bool m_b = (m == b);
bool m_c = (m == c);
bool n_a = (n == a);
bool n_b = (n == b);
bool n_c = (n == c);
bool n_m = (n == m);

{
% Specification
  imp(and(ba,ac),m_a);
  imp(and(ca,ab),m_a);
  imp(and(ab,bc),m_b);
  imp(and(cb,ba),m_b);
  imp(and(ac,cb),m_c);
  imp(and(bc,ca),m_c);
  imp(b_c, m_a);
  imp(c_a, m_b);
  imp(a_b, m_c);

% Implementation
  imp(or(and(ab,bc),and(cb,ba)),
    n_b);
  imp(or(and(ac,cb),and(bc,ca)),
    n_c);
  imp(and(
    not(or(and(ab,bc),and(cb,ba))),
    not(or(and(ac,cb),and(bc,ca)))),
    n_a);

% Violation of the specification
  not(n_m);
}
```

Here `imp(x,y)` denotes that x implies y, and a line starting with `%` is a comment. BoNuS reports a solution in about 1 second on a SUN SPARCstation 5. This is not possible if no search heuristic is used.

We have also experimented with other benchmarks in the literature, such as Triangle, Tomorrow (or Nextdate). The results are satisfactory. Usually the constraint solving time is less than 1 second. However, the inability to deal with arrays prevents us from obtaining more experimental results.

4 Related Work

Constraint logic programming (CLP) [12] is a well-known paradigm for solving Boolean combinations of numeric constraints. Its basic algorithm is goal-reduction: Each time a goal is matched against the head of some clause, and if successful, a new set of subgoals are generated. Over the past 15 years, many CLP systems have been developed, including Prolog III [3], CHIP [4], CLP($\mathcal{R}$) [10], and so on.

In CLP($\mathcal{R}$), linear equations and inequalities are considered to be simple constraints, and they are solved immediately. Non-linear constraints are delayed until they become linear (i.e., when one or more variables get values). During the constraint solving process, the system maintains a set of collected constraints and a set of delayed constraints. As pointed out in [10], linear inequalities are difficult to handle, and eliminating variables from inequalities can be very expensive.

The 2LP framework [13] combines logic programming and linear programming. It can be used as a callable library or a stand-alone system. But it only allows for linear constraints. QUAD-CLP($\mathcal{R}$) [17] can deal with some quadratic constraints (i.e., polynomials of degree 2). Its main idea is to approximate them by linear constraints. It was built on top of CLP($\mathcal{R}$), and it is better than the latter on some non-linear constraint solving problems. But it is still incomplete. Let us consider the constraints in Example 2. Given the following input:

```
X*Y <= 2, 2*X >= Y+3, Y >= 1.2.
```

the system cannot tell us that they are inconsistent. The answer is "Maybe". On the other hand, it can solve some non-linear problems which cannot be solved by our tool. Chan *et al.* [2] combine QUAD-CLP($\mathcal{R}$) with a BDD-based model checker. In their approach, some restrictions are put on the transitions. For example, the assignment $x = x + 1$ is not allowed.

Our tool can be thought of as a specialized constraint solving system. It is an extension of a Boolean satisfiability checker. There is no such operation as unification, which is computationally expensive. Non-linear numeric constraints may be evaluated when some real variables get new bounds, even though no variable gets a specific value.

Compared with the CLP paradigm, our framework is less general. But theoretically, the semantics is simpler and clearer. Practically, the tool might be more efficient on certain kinds of problems. Van Hentenryck pointed out in 1995 (page 566 of [19]):

"Linear programming has been supported in constraint programming languages for about a decade now ... The implementation techniques are however still far from the best techniques used in linear programming packages ... "

We are not sure if the situation has changed during the past 5 years, because many packages are commercial products.

Our framework is also less general than some other paradigms, such as solving first-order constraints in the real closed field. We do not allow such quantifiers as $\exists x \forall y \exists z$ — they are not really needed in our applications. Symbolic computation methods may be more powerful when solving non-linear constraints.

Another more general paradigm is theorem proving. We mentioned the proof checker PVS [16] in Example 4. Heimdahl and Czerny [7] reported some experiences of using it to analyze requirement specifications. PVS has built-in decision procedures for some theories (e.g., Booleans and linear arithmetic). But it is mainly for proving the validity (rather than satisfiability) of formulas. It may not be so efficient as some specialized tools. For instance, when finding the inconsistency between two transitions, the running time of PVS is about 10 seconds [7], while our tool (as well as the BDD approach) needs only a fraction of a second.

Offutt and Pan [14] give a set of heuristic rules for identifying infeasible constraints and equivalent mutants. An example of such a rule is the following one: If $(k_1 > k_2)$, then the two constraints $(x > k_1)$ and $(x < k_2)$ conflict. The arithmetic expressions involved in these rules are very simple. Gotlieb *et al.* [6] also use a constraint solver to generate test data, but real variables are not allowed.

5 Concluding Remarks

Good test generation and analysis tools for software engineering should be able to deal with various data types. In this paper, we describe an extension of a Boolean constraint solver, so that the constraints may involve real variables and enumerated variables. It is simpler than such tools as theorem provers and CLP systems. But it can solve some problems more efficiently.

Our approach focuses on linear constraints. This is not oversimplifying, since they occur frequently in many applications. For example, in the specification of communication protocols, most numeric expressions are linear. In the absence of non-linear constraints, our search algorithm is complete. That is, it will give an answer – satisfiable or unsatisfiable – in a finite amount of time. Thus it can be used to decide the feasibility of selected program paths. In contrast, optimization methods like simulated annealing [18] cannot detect infeasible paths.

We plan to build a full-scale test data generation tool on top of BoNuS. In addition to constraint solving, we need to automate other tasks such as path selection and constraint generation/simplification. Moreover, the solver itself can also be improved. One weakness of the current version is that it cannot deal with arrays directly. We shall study this issue in the future. It is also interesting to investigate other search heuristics, and to extend BoNuS with more powerful mechanisms for dealing with non-linear constraints.

Acknowledgements

The author is grateful to the anonymous reviewers for their helpful comments, to Jimmy Lee and Ho-fung Leung for providing information on CLP systems, to Gilles Pesant for making QUAD-CLP(R) available, and to Qiwen Xu for obtaining some experimental results about PVS. The paper was finished during the author's visit to Ricardo Caferra's group at IMAG, sponsored by the cooperation program between the CNRS of France and the Chinese Academy of Sciences.

References

[1] M. Berkelaar, `lp_solve`, a public domain Mixed Integer Linear Program solver, available from `ftp://ftp.ics.ele.tue.nl/pub/lp_solve/`

[2] W. Chan, R. Anderson, P. Beame, and D. Notkin, "Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints," *Proc. Int'l Conf. on Computer Aided Verification*, pp. 316–327, 1997.

[3] A. Colmerauer, "An introduction to Prolog III," *Comm. of ACM* vol. 33, no. 7, pp. 69–90, 1990.

[4] M. Dincbas *et al.* "The constraint logic programming language CHIP," *Proc. Int'l Conf. on Fifth Generation Computer Systems (FGCS'88)*, 1988.

[5] A. Goldberg, T.C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," *Proc. Int'l Symp. on Software Testing and Analysis*, special issue of *ACM SIGSOFT Software Engineering Notes*, pp. 80–94, 1994.

[6] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," *Proc. Int'l Symp. on Software Testing and Analysis*, *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 2, pp. 53–62, 1998.

[7] M.P.E. Heimdahl and B.J. Czerny, "Using PVS to analyze hierarchical state-based requirements for completeness and consistency," *Proc. IEEE High Assurance Systems Engineering Workshop (HASE'96)*, Oct. 1996.

[8] M.P.E. Heimdahl and N.G. Leveson, "Completeness and consistency analysis of state-based requirements," *Proc. 17th Int'l Conf. Software Eng.*, pp. 3–14, 1995.

[9] M.P.E. Heimdahl and N.G. Leveson, "Completeness and consistency in hierarchical state-based requirements," *IEEE Trans. Software Eng.*, vol. 22, no. 6, pp. 363–377, 1996.

[10] N.C. Heintze, J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap, "The CLP($\mathcal{R}$) programmer's manual," Version 1.2, 1992.

[11] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated consistency checking of requirements specification," *ACM Trans. on Software Eng. and Methodology*, vol. 5, no. 3, pp. 231–261, July 1996.

[12] J. Jaffar and M.J. Maher, "Constraint logic programming: A survey," *J. of Logic Programming* vol. 19/20, pp. 503–581, 1994.

[13] K. McAloon and C. Tretkoff, "2LP: Linear programming and logic programming," *Proc. Int'l Workshop on Principles and Practice of Constraint Programming (PPCP'93)*, pp. 178–189, 1993.

[14] A.J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[15] A.J. Offutt and S. Liu, "Generating test data from SOFL specifications," *J. of Systems and Software*, vol. 49, no. 1, pp. 49–62, 1999.

[16] S. Owre, N. Shankar, and J. Rushby, "PVS: A Prototype Verification System," *Proc. 11th Int'l Conf. on Automated Deduction*, (ed.) D. Kapur, *Lecture Notes in Artif. Intel.*, vol. 607, Springer, pp. 748–752, 1992.

[17] G. Pesant and M. Boyer, "QUAD-CLP(R): Adding the power of quadratic constraints," in: *Proc. 2nd Int'l Workshop on Principles and Practice of Constraint Programming*, (ed.) A. Borning, Springer, pp. 95–108, 1994.

[18] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," *Proc. Int'l Symp. on Software Testing and Analysis, ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 2, pp. 73–81, 1998.

[19] P. Van Hentenryck, "Constraint solving for combinatorial search problems: A tutorial," in: *Proc. 1st Int'l Conf. on Principles and Practice of Constraint Programming (CP'95), Lecture Notes in Comput. Sci.*, vol. 976, Springer, pp. 564–587, 1995.

[20] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a Boolean specification," *IEEE Trans. Software Eng.*, vol. 20, no. 5, pp. 353–363, 1994.

[21] J. Zhang, "A constraint satisfaction tool for hardware design," *Proc. 5th Int'l Conf. on CAD and Computer Graphics*, Shenzhen, China, pp. 517–520, Dec. 1997.

[22] J. Zhang, "Using search techniques in the formal analysis of programs," *Chinese. J. of Advanced Software Research*, vol. 6, no. 2, pp. 169–177, May 1999.