A Dynamic Constraint-Based BMC Strategy For Generating Counterexamples

Hélène Collavizza, Nguyen Le Vinh, Michel Rueher, Samuel Devulder, Thierry Gueguen

A Dynamic Constraint-Based BMC Strategy For Generating Counterexamples*

Hélène Collavizza,
helen@polytech.unice.fr
University of Nice - Sophia
Antipolis, I3S-CNRS, BP 145 06903
Sophia Antipolis Cedex, France

Nguyen Le Vinh,
lvnguyen@polytech.unice.fr
University of Nice - Sophia
Antipolis, I3S-CNRS, BP 145 06903
Sophia Antipolis Cedex, France

Michel Rueher
michel.rueher@gmail.com
University of Nice - Sophia
Antipolis, I3S-CNRS, BP 145 06903
Sophia Antipolis Cedex, France

Samuel Devulder
samuel.devulder@geensoft.com
Geensys, 120 Rue René Descartes,
29280 Plouzané, France

Thierry Gueguen
thierry.gueguen@geensoft.com
Geensoft, 120 Rue René Descartes,
29280 Plouzané, France

ABSTRACT

Checking safety properties is mandatory in the validation process of critical software. When formal verification tools fail to prove some properties, the automatic generation of counterexamples for a given loop depth is achievable, and is therefore an important issue in practice. We propose in this paper a dynamic constraint based exploration strategy for software bounded model checking. Constraint solving is integrated with state exploration to prune state space. Experiments on a real industrial Flasher Manager controller show that our system outperforms state of the art bounded model checking tools.

Keywords

bounded model checking, dynamic exploration strategy, constraint programming, counterexamples, program testing

1. INTRODUCTION

In modern critical systems, software is often the weakest link. Thus, more and more attention is devoted to the software verification process [5]. Software verification includes formal proofs (automatic or semi-automatic), functional and structural testing, manual code review and analysis. In practice, formal proof methods that ensure the absence of all bugs in a design are usually too expensive, or require manual efforts. Thus, automatic generation of counterexamples violating a property on a limited model of the program is an important issue. Typically, it is an open challenge in real time applications where bugs must be found for realistic time periods.

Bounded Model Checking (BMC) techniques have been widely used in semiconductor industry for finding deep bugs in hardware designs [4], and are also applicable for software [14]. In BMC, falsification of a given property is checked for a given *bound*. BMC mainly involves three steps (as described in [15]): (1) the program is unwound k times, (2) the program and the property are translated into a propositional formula ϕ such that ϕ is satisfiable *iff* there exists a counterexample of depth less than k, and (3), a solver (SAT-solver or more recently SMT-solver) is used for checking the satisfiability of ϕ.

In this paper, we propose a dynamic constraint based exploration strategy for BMC of C programs. Instead of translating the program and the property into a big propositional formula and using a constraint solver at the last stage of BMC, constraint solving is integrated with state exploration to prune the state space as early as possible. More precisely, our strategy is based on the following observation: when the program is in an SSA-like form[1], a faulty path can be built in a dynamic way. The Control Flow Graph (CFG) does not have to be explored in a top down (or bottom up) way, and compatible blocks can just be collected in a non-deterministic way. The Dynamic Post-condition-Variable driven Strategy ($DPVS$), takes advantage of this observation. $DPVS$ starts from the post-condition and dynamically collects program blocks which involve variables of the post-condition. Iteratively, it collects blocks which involve the variables used in the blocks where post-condition-variables are defined, and so on. Collecting as much information as possible on a given variable enforces the constraints on its domain and reduces the search space. Thus, inconsistencies are detected as early as possible, and unfeasible

*This work was partially supported by the ANR-07-SESUR-003 project CAVERN and the ANR-07 TLOG 022 project TESTEC.

[1]SSA (Static Single Assignment) form is an intermediate representation used in compiler design: it is a semantics-preserving transformation of a program in which each variable is assigned exactly once [12].

program paths can be cut.

DPVS has been evaluated on a real industrial application, called Flasher Manager, a controller that drives several functions related to the flashing lights of a car. On this real application, *DPVS* outperforms CBMC, a state of the art bounded model checker.

Outline of the paper

Section 2 shows how our approach works on a small example and introduces the new exploration strategy. Section 3 describes the application we used to validate our approach. Section 4 reports experimental results and presents further research directions.

2. DPVS, THE NEW CONSTRAINT BASED SEARCH STRATEGY

In this section, we first describe our approach in very general terms and describe the search process on a small example. Then, we detail the search algorithm.

```
void foo(int a, int b)
1.  int c, d, e, f;
2.  if(a >= 0) {
4.      if(a < 10) {
5.          f = b − 1;
6.      }
7.      else {
8.          f = b − a;
9.      }
10.     c = a;
11.     if(b >= 0) {
12.         d = a; e = b;
13.     }
14.     else {
15.         d = a; e = −b;
16.     }
17. }
18. else {
19.     c = b; d = 1; e = −a;
20.     if(a > b) {
21.         f = b + e + a;
22.     }
23.     else {
24.         f = e ∗ a − b;
25.     }
26. }
27. c = c + d + e;
28. assert(c >= d + e); // property p1
29. assert(f >= −b ∗ e); // property p2
```

Figure 1: Program foo

2.1 Informal Presentation

Consider a program P with a precondition *pred*, a post-condition *post* which is a conjunction of some properties, and a particular property *prop* from *post*. The following pre-processing steps are performed:

1. P is unwound k times (the unwound program is called P_{uw}),
2. P_{uw} is then translated into DSA_{Puw}, its DSA (Dynamic Single Assignment) form [3], where each variable is assigned exactly once on each program path,
3. then DSA_{Puw} is simplified according to the specific property *prop* by applying slicing techniques,
4. the CFG (called G) of the simplified DSA_{Puw} is built,
5. the domains of all variables of G are filtered by propagating constant values along G.

The constraint based dynamic exploration of G works as follows. *DPVS* uses a constraint store S and a queue of variables Q. Q is initialized with the variables in *prop*, whereas S is initialized with the negation of *prop*. As long as Q is not empty, *DPVS* removes the first variable v and searches for a program block where variable v is defined. All new variables (except input variables) of this definition are pushed on Q. The definition of variable v as well as all conditions required to reach the definition of v are added to the constraint store S. If S is inconsistent, *DPVS* backtracks and searches for another definition; otherwise the dual condition to the one added to S is cut off to prevent *DPVS* from losing time in exploring trivially inconsistent paths. When Q is empty, the constraint solver searches for an instantiation of the input variables that violates the property, that's to say a counterexample. If no solution exists, *DPVS* backtracks.

Now, let us illustrate this process on a very small example, the program foo displayed in Figure 1. Program foo has two post-conditions: $p_1 : c >= d + e$ and $p_2 : f > -b * e$. Assume we want to prove property p_1. Figures 2 and 3 depict the paths explored by *DPVS* on the simplified CFG. The search process first selects node (4) where variable c_0 is defined. To reach node (4), the condition in node (0) must be true. Thus, this condition is added to the constraint store S and the other alternative is cut off. At this stage, S contains the following constraints: $\{c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = a_0 \wedge a_0 \geq 0\}$ which can be simplified to $\{a_0 < 0 \wedge a_0 \geq 0\}$. This constraint store is inconsistent and thus *DPVS* selects node (8) where variable c_0 is also defined. To reach node (8), the condition in node (0) must be false. Thus, the negation of this condition is added to the constraint store S and the other alternative is cut off. Now, constraint store S contains the following constraints: $\{c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = b_0 \wedge a_0 < 0 \wedge d_0 = 1 \wedge e_0 = -a_0\}$ which can be simplified to $\{a_0 < 0 \wedge b_0 < 0\}$. This constraint store is consistent and the solver will compute a solution, e.g., $\{a_0 = -1, b_0 = -1\}$. These values of the input variables are a test case which demonstrates that program foo violates property p_1.

This small example illustrates how *DPVS* works. It can also help to understand the intuition behind this new strategy: *DPVS* collects the maximum information on the variables which occur in post-condition to detect inconsistencies as early as possible; this is especially efficient when a small sub-set of the constraint system is inconsistent.

2.2 Algorithm

We now detail algorithm *DPVS* (see algorithm 1).

DPVS selects a variable in Q and tries to find a counterexample with its first definition; if it fails it iteratively tries with the other definitions of the selected variable.

DPVS sets the color of conditional node u to red (resp. blue) when condition of u is set to true (false) in the current

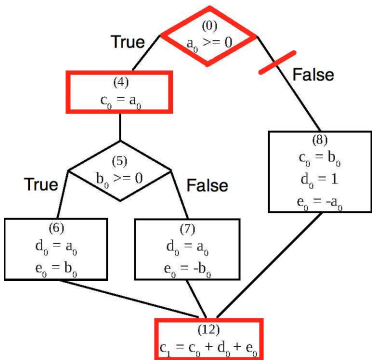

Figure 2: Search process for p_1, step 1

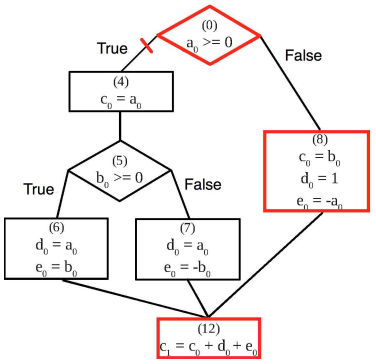

Figure 3: Search process for p_1, step 2

path. In other words, when the color is set to red (resp. blue) the right (resp. left) successor link of u is cut off. $color[u]$ is initialized to blank for all nodes.

DPVS returns *Sol* which is either an instantiation of the input variables of P satisfying constraint system C or $\emptyset$ if C does not have any solution. Solutions are computed by function *solve*, the finite domain solver. Function *solve* is a complete decision procedure over the finite domains. On the contrary, function $isfeasible$, used in line 27, only performs a partial consistency test. In other words, it detects some inconsistencies but not all of them. However, function $isfeasible$ is much more faster than function *solve*; this is the reason why we chose to only perform this test each time the constraints derived from the definition of a variable are added to the constraint store. This partial consistency check can either be done with the finite domain solver (CP) or with the linear programming solver (LP). Of course, the LP solver can only work on a linear relaxation of the constraint system.

It is easy to show that *Sol*, the solution computed by *DPVS* is actually a counterexample. Indeed, these values of the input data satisfy the constraints generated from:

- *pred*, the required precondition;
- $\neg prop$, the negation of a conjunct of the post-condition;
- one definition of all variables in $V(prop)$ and one definition of all variables (except the input variables) introduced by these definitions;
- all conditions required to reach the above mentioned definitions.

Algorithm 1 : *DPVS*

% $du[x]$: set of blocks where variable x is defined

% $anc_c[u]$: set of ancestors of u which are conditional nodes

% $dr[u, v]$: a boolean which is true (resp. false) when the condition of ancestor v of node u has to be true (resp. false) to reach u

% M: set of marked variables (a variable is marked if it has already been put into the queue); M is initialized with $\emptyset$

% S: the constraint store which is initialized with $const(pred \wedge \neg(prop))$ where *const* is a function that transforms an expression in SSA form into a set of constraints

% Q: the set of temporary variables which is initialized with $V(prop)$

Function $DPVS(M, S, Q)$ returns *counterexample*

```
1: if Q = ∅ then
2:    return solve(S)
3: else
4:    x ← POP(Q)
5:    for all u ∈ du[x] do
6:       Cut ← FALSE; SAVE(Q, M, color)
7:       S1 ← S ∧ const(def[x, u])
                {% def[x, u] denotes the definition of x in block u}
8:       V_new ← V(def[x, u]) \ M
9:       PUSH(Q, V_new); add(V_new, M)
10:      for all v ∈ anc_c[u] do
11:         if color[v] = blank then {%no branch is cut off}
12:            V_new ← V(condition[v]) \ M
13:            PUSH(Q, V_new); add(V_new, M)
14:            if dr[u, v] then {% Condition must be true}
15:               S1 ← S1 ∧ cons(condition[v]))
16:               color[v] ← red {% Cut the right branch}
17:            else {% Condition must be false}
18:               S1 ← S1 ∧ ¬ cons(condition[v])
19:               color[v] ← blue {% Cut the left branch}
20:            end if
21:         else
22:            if (color[v] = red ∧ dr[u, v])
                  ∨ (color[v] = blue ∧ ¬(dr[u, v])) then {%no
               branch is reachable}
23:               Cut ← TRUE
24:            end if
25:         end if
26:      end for
27:      if ¬Cut ∧ isfeasible(S1) then
28:         result ← DPVS(M, S1, Q)
29:         if result ≠ ∅ then
30:            return result
31:         end if
32:      end if
33:      RESTORE color, Q, M
34:   end for
35:   return ∅
36: end if
```

Thus, there exists at least one executable path which takes as input values *sol* and computes an output that violates the property *prop*. Otherwise, when no solution can be found, we can state that there does not exist any input values that violate property *prop*; in other words that no counterexample can be found.

3. THE FLASHER MANAGER APPLICATION

In this section we describe the application we used to validate our approach. This real time industrial application comes from a car manufacturer and has been provided by Geensoft[2]. A complete description of this application –with all source code– can be found at `http://users.polytech.unice.fr/~rueher/Benchs/FM/`.
The complexity of this problem is due to the fact that a property must be checked during many stages of execution of the code.

3.1 Description of the module

The *Flasher Manager* is the controller that drives several functions related to the flashing lights of a car. The flashing lights serve several purposes:

1. Under normal operation, when the driver wishes to indicate a direction change, the `CBWS_HAZARD_R` or `CBWS_HAZARD_L` Boolean inputs rise from 0 to 1. The corresponding light (driven by the `CMD_FLASHER_R` or `CMD_FLASHER_L` output respectively) shall then oscillate between an on/off state over a period of 3 time-units (typically 3 seconds). Then, when the input falls back to 0, the corresponding output light shall stop flashing. This is called the `Flashers_left` and `Flashers_right` functions.

2. The driver has the ability to lock and unlock the car from the distance using a RF-key. The state of the open and close buttons of the key is reported to Boolean inputs: `RF_KEY_UNLOCK` and `RF_KEY_LOCK` respectively. The manager has to indicate the state of the doors to the user using the following convention:

   - If the unlock key is pressed while the car is unlocked, nothing shall happen.
   - If the unlock key is pressed when the car is locked, both lights shall flash with a period of 10 time-units during 20 time-units (slow flashes). This is the `Warning_slow` function.
   - If the lock button is pressed while the car is unlocked, both lights shall go on for 10 time-units, and then shall go off.
   - If the lock button is pressed while the car is locked, both lights shall flash during 60 time-units with a period of 1 time-unit (quick flashes for a long time). This is the `Warning_fast` function. It is typically used to locate the car in an over-filled place.

3. Finally the driver has the ability to press the warning button. When a `WARNING` is present (reflected in the value of the `WARNING` input), both lights shall flash with a period of 3 time-units. This is called the `Warning` function.

3.2 Program under test

We have been asked to check the following property (p_1):

The lights should never remain lit

The Simulink model of the *Flasher Manager* has first been translated into a *C* function[3], named f_1. Function f_1 involves 81 Boolean variables including 6 inputs and 2 outputs and 28 integer variables. Function f_1 contains 300 lines of code and mainly consists of nested conditionals including linear operations and constant assignments, as illustrated by the piece of code displayed in Figure 4.

```
and1_a=((Switch5==TRUE)&&(TRUE!=Unit_Delay3_a_DSTATE));
if ((TRUE==((and1_a-Unit_Delay_c_DSTATE)!= 0))) {
   rtb_Switch_b=0;
}
else {
  add_a = (1+Unit_Delay1_b_DSTATE);
  rtb_Switch_b = add_a;
}
superior_a = (rtb_Switch_b>=3);
```

Figure 4: Piece of code of the f_1 function

Property (p_1) of the *Flasher Manager* concerns the behaviour of the *Flasher Manager* for an infinite time period. Practically, we can only check a bounded version of property (p_1): we consider that the property is violated when the lights remain on for N consecutive time periods. We thus introduce a loop (bounded by value N) that counts the number of times where the output of the *Flasher Manager* has consecutively been true. After the loop, if this counter is equal to N, then the property is violated in the sense that the output has remained true for all the period of time. The value of the bound N is fixed as great as possible as shown in section 4; its maximal value is mainly determined by the capabilities of the tools. The part of the C program that corresponds to this bounded version of property (p_1) is displayed in Figure 5.

```
// number of time where the output has been consecutively true
int count = 0;
// consider N periods of time
for(int i=0;i<N;i++) {
   // call to f1 function to compute the outputs
   // according to non deterministic input values
   f1();
   if (Model_Outputs4)
      // the output has been consecutively true one more time
      count++;
   else
      //  the output has not been consecutively true
      count=0;
}
// if count is less than N, then the lights did not remain lit
assert (count<N)};
```

Figure 5: C program under test

4. EXPERIMENTS AND DISCUSSION

In this section, we report and discuss the experiments we have done to validate our approach.

4.1 Tools

DPVS is implemented in `Comet`[4]. There are many re-

[2]See `http://www.geensoft.com/en/`

[3]This translation is done with a proprietary tool of Geensys.

[4]`Comet` is a hybrid optimization platform for solving complex combinatorial optimization problems. `Comet` combines the methodologies used for constraint programming, linear and integer programming, constraint-based local search, and dynamic stochastic combinatorial optimization with a language for modeling and searching (see `http://dynadec.com/technology/`)

strictions on the C programs that the current prototype can handle. Especially, input data are restricted to Booleans, integers and arrays of Booleans and integers. Pointers are not handled[5] and only run-time error-free programs are treated (i.e. errors like dividing by zero or exceptions are not handled).

We compared performances of *DPVS* with CBMC and CPBPV*. CBMC[6] [7] is one of the best bounded model checkers. We used version 3.3.2 that calls the SAT solver *MiniSat*2. CPBPV* is an optimized version of CPBPV [9, 10] which is implemented in `Comet`. Like CPBPV it uses constraint stores to represent both the specification and the program, and to explore execution paths of bounded length over these constraint stores. However, contrary to CPBPV, it works on a simplified CFG. A preliminary bound propagation step is also performed.

Experiments were performed on a Quad-core Intel Xeon X5460 3.16GHz clocked with 16Gb memory. All times are given in seconds. OoM stands for "out of memory" whereas TO stands for "time out" in the different tables. The time out was set to 3 minutes for all benchmarks.

4.2 Experiments on the Flasher Manager

CBMC and *DPVS* found counterexamples that violate the bounded version of property (p_1). They also generated data input sequences such that the flasher lights remain lit for N consecutive periods of time.

For instance, here is a data input sequence that violates this property for 5 time periods:

```
[(0,1,0,0,0,1),(0,1,0,0,0,1),(0,1,0,0,0,1),
(0,0,1,0,0,1),(0,0,0,0,0,1)]
```

where $(0, 1, 0, 0, 0, 1)$ is the value of inputs in_1 to in_6 for the first time period, $(0, 1, 0, 0, 0, 1)$ the value of the inputs for the second time period and so on.

Table 1 shows that *DPVS* outperforms the other approaches on the *Flasher Manager* application. *DPVS* is able to generate counterexamples for instances up to 400 time periods before running out of memory. CPBPV* did not manage to handle this application for instance with n greater or equal than 10.

Table 1: Flasher Manager

N	CBMC	DPVS	CPBPV*
5	0.134	0.026	0,837
10	0.447	0.055	TO
50	12.92	0.345	TO
75	32.74	0.602	TO
100	58.27	2.750	TO
150	138.19	1.552	TO
200	OoM	6.003	TO

4.3 Discussion

Experiments have shown that *DPVS* is very efficient to find counterexamples of property $p1$ of the *Flasher Manager* application. This can be explained by the fact that *DPVS* is a bottom-up dynamic strategy[7]. Since $p1$ does not hold, it is not necessary to explore all the program paths. The challenge is thus to find as early as possible *one* faulty path. Starting from the variables of the precondition gives more chance to early find this faulty path. Furthermore, *DPVS* propagates the most information as possible taking the variables the one after the others, and thus reduces the search space.

We also tested *DPVS* on a well known academic example: the binary search that determines whether a value v is present in a sorted array t. On the contrary to property $p1$ of the *Flasher Manager*, this example is a correct program, thus all program paths have to be explored. Furthermore, this program has a very strong precondition, which seems to recommend a top-down approach.

Table 2 reports the results of the experiments on a correct version of the *Binary Search* program.

Table 2: Binary search (integers of 16 bits)

Length	CBMC	DPVS	CPBPV*
4	5.732	0.529	0.107
8	110.081	35.074	0.298
16	TO	TO	1.149
32	TO	TO	5.357
64	TO	TO	27.714
128	TO	TO	153.646

CBMC and *DPVS* cannot handle this benchmark. CBMC wastes a lot of time in building and exploring the whole formula. The strategy used by *DPVS* is not well adapted for this very specific program. On the contrary, the top-down strategy used in CPBPV* outperforms the other checkers. CPBPV* incrementally adds the decisions taken along a path. This is particularly well adapted for the *Binary Search* program which has a strong precondition. This precondition combined with the decisions taken along a path have a strong impact on feasibility of the next conditions, and help to prune infeasible paths.

Finding an efficient feasibility test is a critical issue: one is face with the trade off of the pruning capabilities versus the speed. We tried different combinations of finite domain constraint solvers and linear programming solvers:

- CP-CP combination: the finite domain constraint solver is used both to check the (partial) consistency at each node and to search a solution;
- LP-CP combination: A linear programming solver is used to check the (partial) consistency of a linear relaxation of the constraint system at each node, and the finite domain constraint solver is used to search a solution.

The reported results concern the CP-CP combination. Using a finite domain solver to check the partial consistency is more efficient on this application than using a LP solver on a linear relaxation of the constraints. Actually, the difference does not come from the efficiency of the solvers itself but from the choice points which are added by the linear relaxation. Let us explain this point on a small example. Consider a test such that $x == y$, the negation of this test corresponds to the constraint $x! = y$ which creates two choice points: $x < y$ and $x > y$.

Furthermore, the domains of the integer variables are small for this application, and the propagation step we perform re-

[5]The prototype used for benchmarks was developed for real time systems with strong programming rules.

[6]see `http://www.cprover.org/cbmc`

[7]It is well known in constraint programming that dynamic strategies are more efficient when only one solution is required.

duces the bounds of the domain. Thus, consistency checks with CP are very efficient.

4.4 Related work

Bounded model checkers transform the program and the post-condition to a big formula and use SAT solvers to prove that the property holds or to find a counterexample [14]. SMT solvers are now used in most of the state of the art BMC tools to directly work on high-level formula (see [2, 15, 11], and the last version of CBMC). Many improvements have been studied for high-level BMC, such as the one proposed in [15], in particular during the unrolling step and to reuse previously learnt lemmas. But to the best of our knowledge, these approaches do not explore the CFG in a dynamic bottom-up approach, that collects non consecutive program blocks.

Constraint Logic Programming (CLP) was used for test generation of programs (e.g., [16, 17, 18, 1]) and provides a nice implementation tool extending symbolic execution techniques [6]. Gotlieb et al showed how to represent imperative programs as constraint logic programs: InKa [16] was a pioneer in the use of CLP for generating test data for C programs. Denmat et al developed TAUPO, a successor of InKa which uses dynamic linear relaxations [13]. It increases the solving capabilities of the solver in the presence of non-linear constraints but the authors only published experimental results for a few academic C programs.

CPBPV [8, 9, 10] is a constraint-based framework for verifying the conformity of a program with its specification under some boundness restrictions. The key idea in CPBPV is to use constraint stores to represent both the specification and the program, and to non-deterministically explore execution paths of bounded length over these constraint stores. CPBPV provides a counterexample when the program is not conforming to the specification. The point is that the search strategies of CPBPV is not well adapted for searching a counterexample. Indeed, CPBPV is based on a top down exploration of the bounded feasible paths because it has been designed for partial program verification. When the goal is only to find a counterexample on a large and complex program, this strategy may become very expensive. In contrast, *DPVS* is a dynamic bottom up strategy which has been designed to find counterexamples on tricky programs.

5. CONCLUSION AND FUTURE WORK

In this paper we have introduced, *DPVS* , a dynamic constraint based strategy for bounded model checking. First experiments with *DPVS* are very encouraging. *DPVS* behaves very well on a non trivial real application. Generating test cases for realistic time periods is a critical issue in real time applications. For the *Flasher Manager* application, *DPVS* generated counterexamples for more significant time periods than CBMC.

These results are impressive since *DPVS* is still a (slow) academic prototype whereas CBMC is a state of the art solver. Of course, other experiments on other applications are required to refine and validate the proposed approach. The dynamic strategy of *DPVS* is very well adapted for problems with a strong post-condition. However, a static top down strategy – like the one used by CPBPV* – is much more efficient for problems with a strong precondition.

Future work also concerns the extension of our prototype. We are working on a new version which handles pointers and which has an interface with a floating point number solver [6], to be able to evaluate the proposed approach on a larger class of programs.

6. REFERENCES

[1] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *LOPSTR 2008*, volume 5438 of *LNCS*, pages 4–23. Springer, 2008.

[2] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, février 2009.

[3] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *Information Processing Letters*, 93(6):281–288, 2005.

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[5] Thomas Bochot, Pierre Virelizier, Hélène Waeselynck, and Virginie Wiels. Model checking flight control systems: The Airbus experience. In *ICSE 2009 (31st International Conference on Software Engineering),Companion Volume*, pages 18–27. IEEE, 2009.

[6] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Softw. Test., Verif. Reliab.*, 16(2):97–121, 2006.

[7] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *TACAS 2004*, volume 2988 of *LNCS*, pages 168–176. Springer-Verlag, 2004.

[8] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS*, volume 3920 of *LNCS*, pages 182–196. Springer, 2006.

[9] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV: A Constraint-Programming Framework for Bounded Program Verification. In *CP 2008*, volume 5202 of *LNCS*, pages 327–341. Springer, 2008.

[10] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV: A Constraint-Programming Framework for Bounded Program Verification. *Constraint*, 15(2):238–264, 2010.

[11] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *ASE*, 0:137–148, 2009.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[13] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. Improving constraint-based testing with dynamic linear relaxations. In *18th ISSRE*, pages 181–190. IEEE Computer Society, 2006.

[14] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[15] Malay K Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In *ICCAD*, pages 794 – 801. ACM, 2006.

[16] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA*, pages 53–62, 1998.

[17] Daniel Jackson and Mandana Vazir. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25. ACM Press, 2000.

[18] Nguyen Tran Sy and Yves Deville. Automatic test data generation for programs with integer and float variables. In *ASE*, pages 13–21. IEEE Computer Society, 2001.