Constraint-Based BMC: A Backward Approach

Hélène Collavizza, Le Vinh Nguyen, Olivier Ponsini, Michel Rueher, Antoine Rollet

Constraint-Based BMC: A Backward Approach*

Hélène Collavizza[1], **Nguyen Le Vinh**[1], **Olivier Ponsini**[1], **Michel Rueher**[1], **Antoine Rollet**[2]

[1] University of Nice – Sophia Antipolis / I3S–CNRS, 2000 route des Lucioles - Les Algorithmes - bât. Euclide B, BP 121 - 06903 Sophia Antipolis Cedex - France, e-mail: lvnguyen@polytech.unice.fr, helen@polytech.unice.fr, michel.rueher@gmail.com, ponsini@i3s.unice.fr,

[2] University of Bordeaux / LABRI–CNRS, 351 Cours de la Libération - 33405 Talence cedex - France, e-mail: rollet@labri.fr

Abstract. Safety property checking 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 therefore an important issue in practice. We investigate in this paper the capabilities of constraint-based bounded model checking for program verification and counterexample generation on real applications. We introduce *DPVS* (Dynamic Post-condition Variable driven Strategy), a new non-sequential search strategy we have developed to handle an industrial application from a car manufacturer, the *Flasher Manager*. This application has been designed and simulated using the *Simulink* platform. However, this module is concretely embedded as a *C* program in a car computer, thus we have to check that the safety properties are preserved on this *C* code.

We report experiments on the *Flasher Manager* application with our constraint-based bounded model checker, and with *CBMC*, a state-of-the-art bounded model checker. Experiments show that *DPVS* and *CBMC* have similar performances on one property of the *Flasher Manager*; *DPVS* outperforms *CBMC* to find a counterexample for two properties; two of the properties of the *Flasher Manager* remain intractable both for *CBMC* and *DPVS*.

Key words: embedded systems, validation, constraint-based bounded model checking, counterexample generation

* Preprint. A preliminary version of this paper appeared in Proceedings of SAC 2011 [11]. This work was partially supported by the ANR-07-SESUR-003 project CAVERN and the ANR-07 TLOG 022 project TESTEC.

1 Introduction

In modern critical systems, software is often the weakest link. Thus, more and more attention is devoted to the software verification process [6]. 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, this 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 [5] and are also applicable for software [16]. In BMC, falsification of a given program property is checked for a given *bound k*. BMC [17] mainly transforms the unwound program and the property into a propositional formula ϕ such that ϕ is satisfiable *iff* there exists a counterexample of depth less than k. A SAT-solver or SMT-solver is used for checking the satisfiability of ϕ.

1.1 Constraint-based bounded model checking

We investigate in this paper the capabilities of constraint-based BMC for program verification and counterexample generation on *real applications*. The goal is to verify the conformity of a program with its specification, that is, to demonstrate that the specification is a consequence of the program under the boundedness restrictions. The key idea in constraint-based BMC 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. The initial constraint store consists of the pre-condition and the negation of the post-condition. The non-deterministic constraint-based symbolic execution incrementally refines this constraint store by adding constraints generated from program conditions and assignments. Like in standard BMC, we assume a bound on the program inputs (e.g., the array length, the size of the integers and the variable values) and on the number of iterations for loops. Boundedness guarantees termination but it may induce incompleteness: indeed, the verifier is inconclusive if executable paths with a length greater than the specified bound exist. The input program is correct if each constraint store produced by the symbolic execution implies the post-condition and the loop unwinding assertion (terminating a loop early) does not fail.

The main difference between constraint-based BMC and standard BMC (i.e., BMC based on SAT or SMT solvers) [16] lies in the representation of the program and the assertions: the standard BMC approach generates a big Boolean formula whereas we generate the constraints on the fly. We proposed in a previous work a constraint-based BMC framework named *CPBPV* [9,10]. *CPBPV* is based on a depth first search strategy that explores the Control Flow Graph (CFG) of a program starting from the pre-condition. *CPBPV* incrementally adds the constraints associated to the nodes of the CFG, pruning unfeasible paths as early as possible. *CPBPV* has been successful on classical benchmarks, such as sorting algorithms or the binary search algorithm. However, *CPBPV* was unable to prove or to disprove any of the properties of the *Flasher Manager* application. Therefore, we investigated new search strategies and designed *DPVS* (Dynamic Post-condition Variable driven Strategy), a non-sequential bottom-up search strategy. *DPVS* starts from the post-condition and collects as much information on the variables of the post-condition as possible, to detect inconsistencies as early as possible.

The contribution of this paper is twofold:

i) We provide a new industrial application which is still a challenge for state-of-the-art bounded model checkers. While BMC has widely been applied on device drivers, as illustrated by the very successful SLAM project [3], this application comes from the automotive industry. This class of application is generally simulated or verified using platforms such as *Simulink* or *SCADE*, but we are interested in the correction of the *C* code which is actually embedded in the car computer[1]. All the source code of this new kind of application is publicly available.

ii) We introduce *DPVS*: a non-sequential bottom-up search strategy. It behaves well on this application: *DPVS* and *CBMC* have similar performances on three properties of the *Flasher Manager* but *DPVS* outperforms *CBMC* to find a counterexample for two other properties.

[1] Otherwise, the compiler from *Simulink* to *C* should be formally verified.

1.2 Outline of the paper

We first recall the basics on constraint-based bounded model checking. Then, we introduce the search strategy we have developed to handle the *Flasher Manager* application. Next, we describe the *Flasher Manager* application, a real time industrial application from a car manufacturer. We describe the *Simulink* module of the *Flasher Manager* and the properties we have to check. We also explain how the *Simulink* module and the properties have been translated into *C* programs which are used as inputs to the bounded model checkers. Finally, we report experiments on the *Flasher Manager* application with *DPVS* and with *CBMC*.

2 Constraint-based BMC

We first recall some basics on Constraint Programming (CP) techniques [25] that are useful to understand the rest of the paper; the reader familiar with CP techniques can skip the next subsection.

In a second part, we introduce constraint-based BMC and the *DPVS* strategy.

2.1 Basics on CP

Constraint Programming (CP) is a *way of modeling and solving combinatorial optimization problems.* CP combines techniques from artificial intelligence, logic programming, and operations research. Several industrial solvers (e.g., `ILOG/IBM`[2], `Comet`[3]) and academic solvers (e.g., `Gecode`[4], `Choco`[5], `Minion`[6]) are available. There are many successful industrial applications, e.g., timetabling (Dutch railway), hardware verification (Intel), scheduling, planning (see [25], Part II).

The key features of CP are:

- *Domain filtering* that considers each constraint separately and *removes values that are trivially inconsistent*;
- *Search strategies* that try to exploit the *structure of the problem*;
- *Global constraints* that use (efficient) specific polynomial algorithms for some subclasses of constraints.

To illustrate the intuition behind "domain filtering" consider the following example:

[2] `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`

[3] `http://www.comet-online.org`

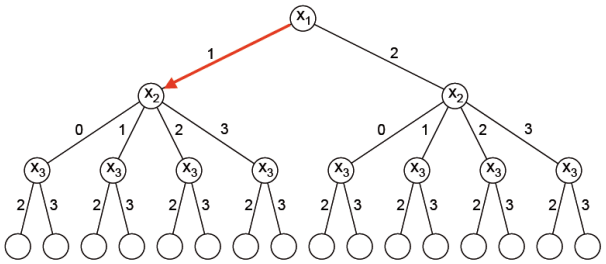
[4] `http://www.gecode.org`

[5] `http://choco.sourceforge.net/api/choco/Solver.html`

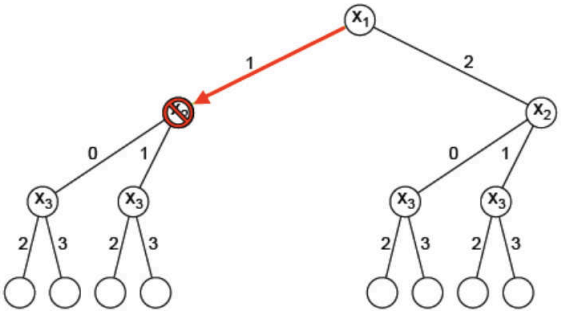
[6] `http://minion.sourceforge.net`

– variables/domains:
  $x_1 \in \{1,2\}$, $x_2 \in \{0,1,2,3\}$, $x_3 \in \{2,3\}$;
– constraints:
  $\{x_1 > x_2,\ x_1 + x_2 = x_3,\ alldifferent(x_1, x_2, x_3)\}$.

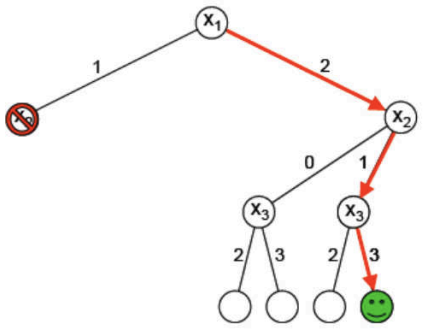
A naive search process would explore the following tree, starting by assigning 1 to variable x_1:



However, if we just consider constraint $x_1 > x_2$, we can remove values $\{2,3\}$ from the domain of variable x_2. That is what domain filtering does in the first step. Then, the search process starts on the following reduced search tree:



Value 1 is assigned to variable x_1 and the filtering process is restarted. Now, if we consider constraint $x_1 + x_2 = x_3$ the domains of variables x_2 and x_3 become empty. So, we will backtrack and assign value 2 to variable x_2. The filtering process will use again constraint $x_1 + x_2 = x_3$ to remove value 0 from the domain of variable x_2 and the global constraint *alldifferent* to remove value 2 from the domain of variable x_3. Thus, we only needed two assignments to find the unique solution. The final search tree is:



Global Constraints are very important to solve difficult problems. The *alldifferent* global constraint [24] is probably one of the most famous global constraint. It uses matching algorithms for bipartite graph and network flow algorithms to handle a set of constraints of the form $x_i \neq x_j$. In program verification, we can use a linear programming solver (LP) as a global constraint for handling a set of linear constraints.

The solver efficiency also highly depends on search strategies. These strategies use heuristics for choosing the variable to instantiate and for choosing the value for the selected variable. *First fail* is one of the most popular heuristic. The intuition behind *First fail* is: "to succeed, try first where you are most likely to fail".

2.2 CP framework & BMC: overview

The goal of BMC is to mechanically check properties of models. In BMC, the falsification of a given property is checked for a given bound k. BMC mainly involves three steps:

1. The program is unwound k times.
2. The unwound program and the property are translated into *a big propositional formula* ϕ. ϕ is satisfiable *iff* there exists a counterexample of depth less than k.
3. A SAT-solver or SMT-solver is used for checking the satisfiability of ϕ.

In our CP framework, the falsification of a given property is also checked for a given bound and also involves three steps:

1. The program is unwound k times.
2. An annotated and simplified *CFG* is built.
3. This CFG and the specification are translated into constraints *on the fly*. Unfeasible paths are *pruned as early as possible*.

The main difference between the two approaches are:

– Standard BMC builds a big propositional formula whereas CP-based BMC generates constraints on the fly, according to the exploration strategy of the CFG.
– Standard BMC uses efficient SAT-solvers (or more recently SMT-solvers) whereas CP-based BMC uses a set of solvers, some of which are well suited for handling numeric expressions.

2.3 CP framework & BMC: details

More precisely, consider a program P with pre-condition *pre*, post-condition *post* which is a conjunction of some properties, and a particular property *prop* from *post*[7]. The following pre-processing steps are first performed:

1. P is unwound k times yielding program P_{uw};

[7] *post* is a conjunction, so its negation (which is added into the constraint system) is a disjunction. Practically, it is often more efficient to handle each conjunct individually.

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

Then G is explored and a constraint system is generated on the fly as follows:

1. A constraint system CS is created and initialized with the constraints associated with the pre-condition *pre* and the negation of the property to be proven $\neg prop$.
2. Nodes of G are translated and added to CS on the fly. If the current node n is non-conditional, then it is simply translated as a constraint and added to CS. If n is a conditional node, then its condition is temporarily added to CS to *check* its feasibility. If it is not consistent with CS, the corresponding path is cut.
3. When the end of an execution path has been reached, CS is *solved*. If it has a solution, it is a counterexample, since it satisfies the constraints of the pre-condition, of the program path and the negation of the post-condition. If CS has no solution, the program is correct along this path (under the boundedness hypothesis).

To increase performances, our constraint framework is parametrized with a list of solvers which are tried in sequence, starting with the least expensive and the least general one. Our prototype implementation uses a linear programing solver, a mixed integer-programming system and a finite domain constraint solver. The feasibility check in the above step 2 is only a partial consistency test: it may not detect some inconsistencies. In contrast, in step 3, the constraint system solving relies on a complete decision procedure.

As we already mentioned, the main difference with standard BMC that uses a SAT or SMT solver is that our constraint-based BMC approach builds the constraint system *on the fly* during CFG exploration. A major lever to improve our method is therefore the way the CFG is explored. Thus, we developed some *search strategies* that best meet these two objectives:

1. cut unfeasible paths as early as possible;
2. when there is a counterexample, explore the faulty path as early as possible.

The first strategy we have developed was a naive depth first search strategy, called *CPBPV* [10]. This strategy was successful on academic benchmarks, in particular for programs with a strong pre-condition (e.g., the binary search program). However, this strategy fails on the *Flasher Manager* application because it has no pre-condition and its main function manages many functionalities while the properties to be proven cover a small subset of these functionalities (see discussion in Sect. 5).

We have thus developed the *DPVS* search strategy, that takes benefits of the post-condition to explore the CFG in a bottom-up way. This is detailed in the next subsection.

2.4 DPVS: a non-sequential search strategy

Contrary to *CPBPV*, *DPVS* is a *non-sequential bottom-up strategy* which has been designed to find counterexamples for program properties. In *DPVS*, the exploration of G is guided by the variables. The first explored node is the post-condition. The next explored nodes are the nodes that define the variables involved in the post-condition, and so on. At each step, the explored node defines a variable directly involved in the post-condition or a variable that is indirectly involved in the post-condition. Note that the nodes are explored in a non-sequential way: a variable involved in an expression of a node of depth n may be defined in a node of depth $n-i$.

This strategy is based on the following observation: when the program is in an SSA-like form[8], a faulty path can be built in a dynamic way. In other words, the CFG does not have to be explored in a top-down (or bottom-up) sequential way, and compatible blocks can just be collected in a non-deterministic way.

Next section explains how DPVS works in an informal way.

2.4.1 Informal presentation

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* (written $V(prop)$ in the following), S is initialized with *pre* and the negation of *prop*. As long as Q is not empty, *DPVS* dequeues the first variable v and searches for a program block where variable v is defined. All new variables (except input variables) of the definition of variable v are enqueued 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 is 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 Fig. 1. Program

[8] SSA (Static Single Assignment) form is an intermediate representation used in compilation: it is a semantics-preserving transformation of a program in which each variable is assigned exactly once [14].

```
1  void foo(int a, int b)
2    int c, d, e, f;
3    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
30 }
```

Fig. 1. Program foo

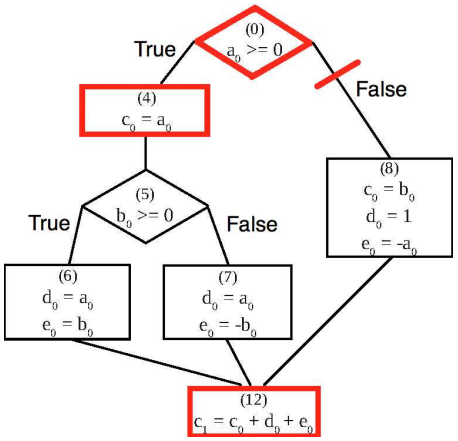


Fig. 2. Simplified CFG for p_1, step 1

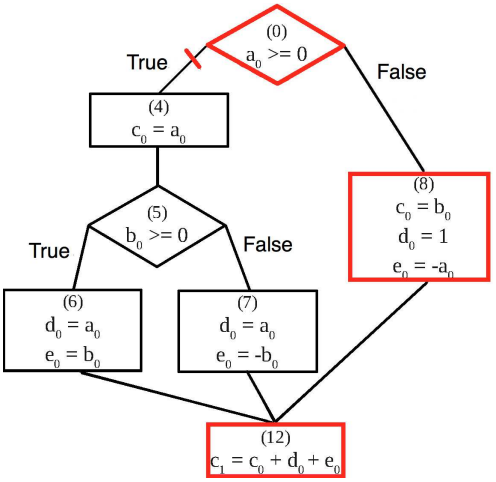


Fig. 3. Simplified CFG for p_1, step 2

`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 (i.e., the CFG where slicing techniques have removed lines 4 to 9 and 20 to 25 which have no impact on p_1). At this step, S is equal to $c_1 < d_0 + e_0$. The search process selects node (4)[9] where variable c_0 is defined (for simplicity reasons, we omit the first step where c_1 is treated, and c_0 is trivially added to Q; see Algorithm 1 for more details). To reach node (4), the condition in node $\langle 0 \rangle$ must be true. Thus, this condition is added to the constraint store S and the other alternative ($a_0 < 0$) is cut off. At this stage (see Fig. 2), 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 $\langle 0 \rangle$ must be false. Thus, the negation of this condition is added to the constraint store S and the other alternative is cut off. Now (see Fig. 3), 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 that demonstrates that program *foo* violates property p_1. *DPVS* was able to find this counterexample without ever visiting nodes (5), (6) and (7).

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

Next subsection details algorithm *DPVS*.

2.4.2 Algorithm

Algorithm *DPVS* (see Algorithm 1) is called with four parameters:

[9] Notation (n) will be used for statement nodes, and $\langle n \rangle$ for conditional nodes in the CFG of Fig. 2 and 3.

- S: the constraint store; S is initialized with $cst(pre \wedge \neg(prop))$ where cst is a function that transforms an expression in DSA form into a set of constraints.
- Q: the LIFO queue of temporary variables; Q is initialized with $V(prop)$.
- M: the set of marked variables (a variable is marked if it has already been put into the queue); M is initialized with $V(prop)$ –the variables of $prop$– and the input variables of the function.
- $color[v]$: the color of a node, used to mark the paths. A node is *Blank* when it has not been explored, *Red* when it has been explored coming from left and *Blue* coming from right.

Algorithm *DPVS* also uses the following data structures:

- $du[x]$: the set of blocks where variable x is defined.
- $anc_c[u]$: the set of ancestors of u that are conditional nodes whose condition value determines the reachability of u.
- $dr[u, v]$: 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.

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 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 an instantiation of the input variables of P satisfying constraint system S or it returns $\emptyset$ if S does not have any solution. Solutions are computed by function *solve* (line 28), using the finite domain solver (CP). Function *solve* is a complete decision procedure over the finite domains. On the contrary, function *isfeasible* used in line 33 only performs a partial consistency test. In other words, it detects some inconsistencies but not all of them. However, function *isfeasible* is much faster than function *solve*; this is the reason why we chose to perform only 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. Thus, the *solve* function is called only once when the end of a path has been reached.

2.4.3 Soundness of Algorithm 1

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:

Algorithm 1 : *DPVS*

Function $DPVS(S, Q, M, color)$ returns a counterexample

Require: $Q \neq \emptyset$
1: $x \leftarrow$ POP(Q)
2: **for all** $u \in du[x]$ **do**
3: $Cut \leftarrow$ **false**; SAVE($S, Q, M, color$)
4: $S_1 \leftarrow S \wedge\ cst(def[x, u])$
{% $def[x, u]$ denotes the definition of x in block u}
5: $V_{new} \leftarrow V(def[x, u]) \setminus M$
6: PUSH(Q, V_{new}); add(V_{new}, M)
7: **for all** $v \in anc_c[u]$ **do**
8: **if** $\neg Cut$ **then**
9: **if** $color[v] = Blank$ **then** {%first visit of v}
10: $V_{new} \leftarrow V(condition[v]) \setminus M$
11: PUSH(Q, V_{new}); add(V_{new}, M)
12: **if** $dr[u, v]$ **then** {% Condition must be true}
13: $S_1 \leftarrow S_1 \wedge\ cst(condition[v]))$
14: $color[v] \leftarrow Red$ {% Cut the right branch}
15: **else** {% Condition must be false}
16: $S_1 \leftarrow S_1 \wedge \neg\ cst(condition[v])$
17: $color[v] \leftarrow Blue$ {% Cut the left branch}
18: **end if**
19: **else**
20: **if** $(color[v] = Red \wedge \neg dr[u, v])$ $\vee\ (color[v] = Blue \wedge dr[u, v])$ **then** {%no branch is reachable}
21: $Cut \leftarrow$ **true**
22: **end if**
23: **end if**
24: **end if**
25: **end for**
26: **if** $\neg Cut$ **then**
27: **if** $Q = \emptyset$ **then** {% end of a path }
28: $Sol \leftarrow$ solve(S_1)
29: **if** $Sol \neq \emptyset$ **then**
30: **return** Sol {% a counterexample has been found}
31: **end if**
32: **else**
33: **if** $isfeasible(S_1)$ **then** {% current path is feasible, so recursive call}
34: $Sol \leftarrow DPVS(S_1, Q, M, color)$
35: **if** $Sol \neq \emptyset$ **then**
36: **return** Sol {% a counterexample has been found}
37: **end if**
38: **end if**
39: **end if**
40: **end if**
41: RESTORE($color, M, Q, S$)
42: **end for**
43: **return** $\emptyset$

– *pre*, the required pre-condition;
– $\neg prop$, 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.

Thus, there exists at least one executable path which takes as input values *Sol* and computes an output that violates the property *prop*.

Conversely, if there exists a counterexample violating *prop*, then there exists at least one executable path in the CFG corresponding to this case. Algorithm 1 guaranties that:

– any node with a definition of a variable involved in *prop* is explored (with the corresponding constraints added to S);
– any conditional node ancestor of an explored node is considered with the appropriate predicate added to S;
– any node with a definition of a variable used in an explored node is also explored (with the corresponding constraints added to S).

Consequently, a consistent constraint system S corresponding to the faulty path has been built and solved by Algorithm 1. The solution of this constraint system provides a counterexample of *prop*.

Otherwise, when no solution can be found, we can state that there does not exist any input values violating property *prop*; in other words, no counterexample can be found with the boundedness hypothesis.

3 The *Flasher Manager* application

In this section we describe the *Flasher Manager* application.

This real time industrial application from a car manufacturer has been provided by `Geensoft / Dassault Systems`[10]. The *Flasher Manager* application was designed and simulated using the *Simulink* platform. Its specification was given by Geensoft and consists of four main properties. The *Flasher Manager* is concretely embedded as a *C* program in a car computer, thus our aim is to check that the four properties are preserved on this *C* program. This is a challenging software verification problem: its complexity comes from the size of the *C* function generated from the *Simulink* module and from the number of clock cycles required for verifying each property.

We first describe the *Simulink* module of the *Flasher Manager*, then the properties we have to check and last, we explain how the *Simulink* module and the properties have been translated as a *C* program in order to use a bounded model checker for the verification. A description of this application –with all source code– can be found at `http://users.polytech.unice.fr/~rueher/Benchs/FM`.

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

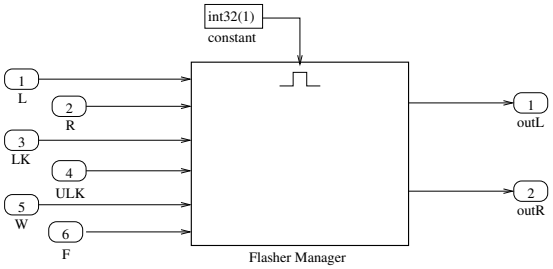


Fig. 4. Simplified *Simulink* model of the *Flasher Manager*

3.1 Description of the Simulink module

The *Flasher Manager* is a controller that drives several functions related to the flashing lights of a car. Each function is enabled by some input commands, activates one or two flashing lights, and is described by its duration and its flashing period (i.e., time-units required to oscillate from 1 to 0 or 0 to 1). The next subsections detail these points for the three main functionalities of the *Flasher Manager*. Figure 4 shows a simplified *Simulink* model (i.e., inputs/outputs) and Fig. 5 provides a more detailed model.

3.1.1 Direction change

When the driver indicates a direction change, Boolean input `R` or `L` rises from 0 to 1. The corresponding light (respectively driven by the `outR` or `outL` output) then oscillates between on/off states with a period of 6 time-units (typically 3 seconds). Thus, an output sequence of the form [111000] is repeated on one of the lights. Then, when the input falls back to 0, the corresponding output light stops flashing. The light starts oscillating immediately when the command is enabled, and stops immediately when the command is disabled. These are the `Flashers_left` and `Flashers_right` functions.

3.1.2 Lock and unlock of the car

The driver has the ability to lock and unlock the car from the distance using an RF-key. The state of the unlock and lock buttons of the key is reported to Boolean inputs `ULK` and `LK` respectively.

When an RF-key is pressed, the manager indicates the state of the doors to the user using the following convention:

– If the unlock button is pressed while the car is unlocked, nothing shall happen.

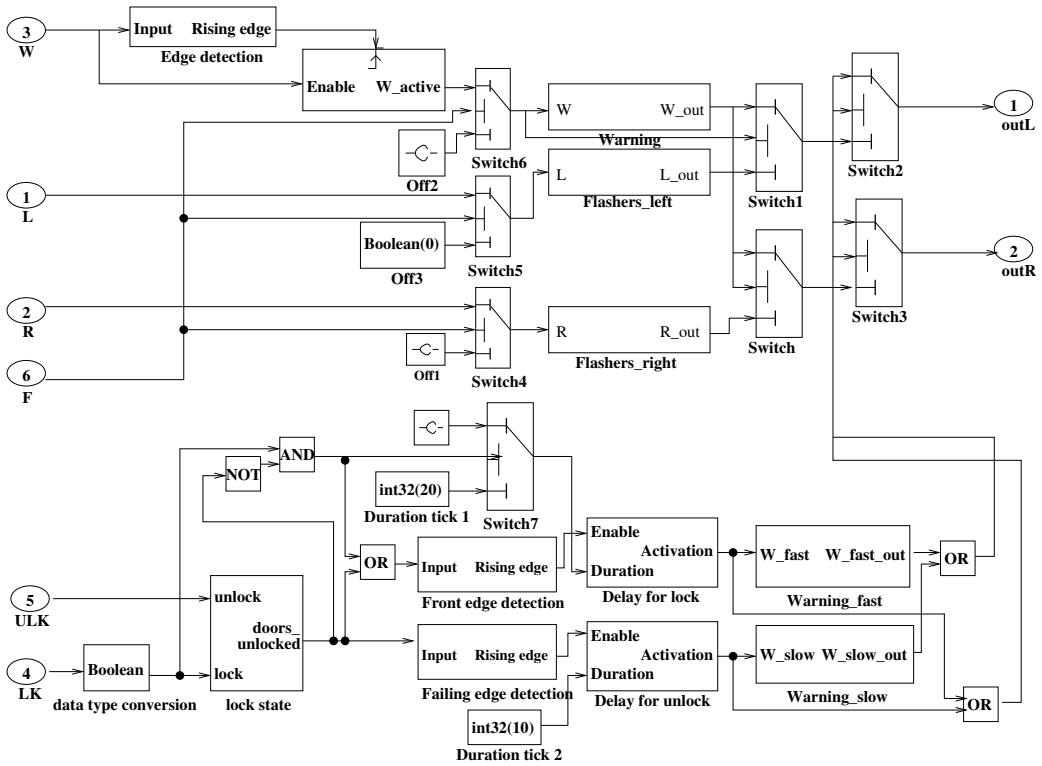


Fig. 5. Detailed *Simulink* model of the *Flasher Manager*

- If the unlock button is pressed while the car is locked, both lights shall flash with a period of 2 time-units during 20 time-units (fast flashes for a short time). More precisely, when the `ULK` input is activated, the oscillation starts two cycles after the activation, produces an output sequence of the form [10101...010] on both lights and stops on the 22^{nd} cycle after the activation. This is the `Unlock_flash` 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 for another 10 time-units, producing an output sequence of the form [1111111110000000000] on both lights.
- If the lock button is pressed while the car is locked, both lights shall flash during 60 time-units with a period of 2 time-units (fast flashes for a long time). More precisely, when the `LK` input is activated, the oscillation starts the next cycle, produces an output sequence of the form [10101...010] on both lights that stops on the 61^{st} cycle. This is the `Lock_flash` function. It is typically used to locate the car in an overfilled place.

Note that in the initial state, the doors are locked.

3.1.3 Warning function

Finally, the driver has the ability to press the warning button. When the warning is on, both lights flash with a period of 6 time-units (slow flashes). This is the `Warning` function. The `W` input is a *push-down* button. In the initial state of the manager, the warning is off. A rising edge of `W` activates the warning and the next rising edge of `W` deactivates the warning.

3.2 Properties

We checked four properties of the *Flasher Manager* module.

Property 1: "`Warning` function has priority over other flashing functions."

Property 2: "When the warning button has been pushed and then released, the `Warning` function resumes to the `Flashers_left` (or `Flashers_right`) function, if this function was active when the warning button was pushed."

Property 3: "When the `F` signal (for flasher active) is off, then the `Flashers_left`, `Flashers_right` and `Warning` functions are disabled. On the contrary, all the functions related to the lock and unlock of the car are maintained."

Property 4: "Lights should never remain lit infinitely."

Restrictions For checking the four properties, we will adopt the following restrictions in accordance with the designers of the *Flasher Manager* module:

1. `L` and `R` inputs cannot both be `TRUE` on the same cycle;
2. `LK` and `ULK` inputs cannot both be `TRUE` on the same cycle.

```
1  and1_a=((Switch5==TRUE)
2         &&(TRUE!=Unit_Delay3_a_DSTATE));
3  if ((TRUE==((and1_a-Unit_Delay_c_DSTATE)!= 0))) {
4     rtb_Switch_b=0;
5  }
6  else {
7    add_a = (1+Unit_Delay1_b_DSTATE);
8    rtb_Switch_b = add_a;
9  }
10 superior_a = (rtb_Switch_b>=3);
```

Fig. 6. Piece of code of the `f1` function

First restriction means that we do not consider a degraded use when the lever of the indicators is damaged. Second restriction excludes a misuse of the RF-key.

The next subsection describes how the four properties have been modeled as C programs in order to be checked with bounded model checkers.

3.3 Programs under verification

We first describe the C function built from the *Simulink* model, then we give the general principles to model the properties as C programs using this function, and last we detail each property.

3.3.1 The `f1` function

The *Simulink* model of the *Flasher Manager* was automatically translated into a C function, named `f1`, using a Geensoft proprietary tool. This function `f1`, together with the scheme of the *Simulink* model and an informal textual description of the properties of the module, is the material that was provided to us by the designers. Function `f1` involves 81 Boolean variables including 6 inputs and 2 outputs and 28 integer variables. It 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 Fig. 6.

A call to the `f1` function simulates one cycle of the *Simulink* module: state variables and output variables are modified according to the values of the input variables.

3.3.2 Modeling for BMC

In order to verify the properties using a BMC tool, we associate a C program to each property. Such a program starts with a call to the function that initializes the module, and then mainly consists of a loop that:

1. reads the inputs,
2. calls the `f1` function,
3. states some assertions on the outputs; these assertions may depend on current and previous values.

In step 1, unknown input values are represented by a call to function `nondet_in()`. This is the syntax required by the BMC tools we use. Statement `int v=nondet_in()` is translated by our BMC tools as the creation of a variable `v` with an initial domain that contains the whole set of machine integers (the integer format can be set as an option of the BMC tools). In step 3, we use some `assert` statements which are checked in sequence by the BMC tools.

For each property, the loop must be checked for a number of iterations that is tractable, but nevertheless meaningful. BMC tools make a *bounded* verification. Generally, the bound is progressively increased until an error is found, or the size of the formula is too large and exceeds the capacity of the tool. For the *Flasher Manager* module, the longest function is the `Lock_flash` function: it requires 61 cycles. However, in our experiments (see Sect. 4), properties often require much more cycles.

3.3.3 Property 1

To check the property 1, we have to show that whatever input commands are, if the warning is on, the lights must flash as described by the `Warning` function: both lights shall oscillate with a time period of 6 time-units. The C program used to model this property (called `prop1`) is shown in Fig. 7:

- Each iteration of the main loop (line 8) is a clock cycle where input values are read (lines 9 to 15) and `f1` function is called (line 25). Note that lines 11 and 13 apply the restrictions mentioned in Sect. 3.2.
- The command of the warning is a push-down button. Thus the variable `Won` in the program becomes `TRUE` only when it was `FALSE` before, and there is a rising edge of the input `W` (lines 19 to 22).
- If `Won` is `TRUE`, then the lights shall oscillate with a time period of 6 time-units, producing an output sequence of the form [111000111000...]. We use a variable `count` that counts the number of cycles since `Won` is true. Value of variable `count` is set to 0 on each rising edge of `W` (line 20), and is increased when `Won` is true (line 34). `count/3` brings the oscillation back to a one time-unit oscillation. Since the oscillation starts with 1, the lights shall be `TRUE` when `(count/3)%2==0` and shall be `FALSE` when `(count/3)%2==1`. The `assert` statements at lines 31 and 33 are stating this.

3.3.4 Property 2

Property 2 concerns a scenario that hosts three successive events: a direction change, a warning activation, and a warning deactivation. Since Property 1 is false (because lock and unlock functions have priority on the warning), we set the entries `LK` and `ULK` to `FALSE` to simplify the modeling[11].

[11] Experimental results (see Sect. 4.2.2) show that even this simplified version is not tractable.

```
1  void prop1(int d) {
2    init(); f1() // module initialization
3    F=TRUE;       // the flashing function is active
4    _Bool oldW; // old value of warning input
5    _Bool Won = FALSE; // true if warning function is on
6    int count=0;// number of cycles when Won was TRUE
7    // consider d time-units
8    for (int i=0;i<d;i++) {
9     // non-deterministic values of the inputs
10     L=nondet_in();
11     if (L) R=FALSE; else R=nondet_in();
12     LK=nondet_in();
13     if (LK) ULK=FALSE; else ULK=nondet_in();
14     oldW=W;
15     W=nondet_in();
16     // warning is a push-down button
17     // Won is TRUE if it was FALSE before
18     // and there has been a rising edge of input W
19     if (W && !oldW){
20        count=0;
21        Won = !Won;
22     }
23     // call to f1() to simulate one pass through
24     // the module
25     f1();
26     // if the warning function is on, the flashing
27     // lights shall oscillate with a period of
28     // 6 time-units, starting with value 1
29     if (Won) {
30      if ((count/3)%2==0)
31        assert(outL==TRUE && outR==TRUE);
32      else
33        assert(outL==FALSE && outR==FALSE);
34      count++;
35     }
36   }
37 }
```

Fig. 7. Function under verification for Property 1

The C program associated with property 2 (called `prop2` and shown in Fig. 8), works as follows:

- It uses counter `countR` (resp. `countL`) to count the number of cycles since `R` (resp. `L`) has been `TRUE`. Note that `countL` has the initial value -2 because a left direction change activates the left light on the next cycle of the rising edge of `L`.
- The first loop (line 11) is used to repeat the scenario.
- When the warning is activated, and a right or left direction change is also active (line 22), then another loop starts and stops when the warning is cut down (line 24).
- When the warning has been cut and a right or left direction change is active, the lights shall behave as the `Flashers_left` or `Flashers_right` functions (lines 38 to 49).

```
1  void prop2(int d) {
2   init();
3   // LK and ULK are not active
4   LK=FALSE; ULK=FALSE;
5   // the flashing function is active
6   F=TRUE;
7   _Bool oldW; // old value of W input
8   _Bool Won = FALSE; // true if warning function is on
9   int countR=-1; // number of cycles since R was TRUE
10  int countL=-2; // number of cycles since L was TRUE
11  for (int i=0;i<d;i++) {
12    // read non deterministic inputs with restrictions
13    L=nondet_in();
14    if (L) R=FALSE; else R=nondet_in();
15    if (R) countR++; else countR=-1;
16    if (L) countL++; else countL=-2;
17    oldW=W;   W=nondet_in();
18    if (W && !oldW) Won = !Won;
19    // call to the module
20    f1();
21    // R or L has been activated later (or equal) than W
22    if ((Won && R) || (Won && L)) {
23      // read inputs until the warning is released
24      for (int j=0;j<d && Won;j++) {
25         L=nondet_in();
26         if (L) R=FALSE; else R=nondet_in();
27         if (R) countR++; else countR=-1;
28         if (L) countL++; else countL=-2;
29         oldW=W;   W=nondet_in();
30         if (W && !oldW) Won = !Won;
31         // call to the module
32         f1();
33    }
34    // if the warning function is released
35    // and a direction change is still active,
36    // the flasher_right or flasher_left function is
37    // active
38    if (!Won && R) {
39      if ((countR/3)%2==0)
40         assert(outL==FALSE && outR==TRUE);
41      else
42         assert(outL==FALSE && outR==FALSE);
43    }
44    if (!Won && L) {
45       if (countL==-1 || (countL/3)%2==0)
46          assert(outL==TRUE && outR==FALSE);
47       else
48          assert(outL==FALSE && outR==FALSE);
49    }
50  }
51 }
```

Fig. 8. Function under verification for Property 2

```
1  void prop3b(int d) {
2    init();
3    // LK and ULK are not active
4    LK=FALSE;
5    ULK=FALSE;
6    for (int i=0;i<d;i++) {
7      // read non deterministic inputs with restrictions
8      F=nondet_in();
9      L=nondet_in();
10     if (L) R=FALSE; else R=nondet_in();
11     W=nondet_in();
12     f1();
13     // if F is disabled, left, right and
14     // warning functions are deactivated
15     if (!F) {
16       assert(outL==FALSE && outR==FALSE);
17     }
18   }
19 }
```

Fig. 9. Function under verification for Property 3b: left, right and warning flashing functions

3.3.5 Property 3

Property 3 concerns the behavior of the module when the flashing function is deactivated, i.e., input signal `F` is `FALSE`. We wrote two versions of this property: Properties 3a and 3b. Property 3a corresponds to the original property which states that `F=FALSE` deactivates the `Flashers_left`, `Flashers_right` and `Warning` functions, but does not deactivate the functions related to lock and unlock commands. However, we restricted `LK` and `ULK` inputs so that they are ignored while a flashing function due to a previous lock or unlock of the car is not yet completed. Even with this restriction on the combinations of `LK` and `ULK` inputs, Property 3a is not tractable because of combinatorial explosion (see Sect. 4.2.3). Hence, we introduced Property 3b which only deals with `Warning`, `Flashers_left` and `Flashers_right` functions: unlock and lock inputs are disabled. We first describe Property 3b, as it is a simplification of Property 3a; next, we detail Property 3a.

The *C* program associated with Property 3b –called `prop3b`– is shown in Fig. 9. It mainly consists in a loop where inputs are read (lines 7 to 12), `f1` function is called, and lights are checked to be off each time `F` is equal to `FALSE` (line 16).

The *C* program associated with Property 3a –called `prop3a`– is shown in Fig. 10 and 11. The main difficulty to model this property is that there are four possible scenarios, which depend on the `LK` and `ULK` inputs, and also on the state of the doors (closed or not). Furthermore, combination of functions related to lock/unlock must be ignored, thus one has to record if a previous function is now complete. `prop3a` works as follows:

- Lines 3 to 14 initialize the state variables related to lock and unlock functions.
- Lines 17 to 19 read values of inputs `F`, `L`, `R` and `W`.
- Lines 20 to 40 read values of inputs `LK` and `ULK`, only if no lock/unlock function is active; state variables related to `LK` and `ULK` are also updated.
- Assertions must only be checked when `F` is `FALSE` (line 46).
- Lines 47 to 49 concern the case where no function related to lock or unlock is active. Thus the lights shall be off.
- Lines 50 to 64 concern the case where a function related to lock is active. The assertion to be checked depends on the state of the doors (opened or not) when the lock was activated.
- Lines 65 to 76 concern the case where a function related to unlock is active. Here also, the assertion to be checked depends on the state of the doors (opened or not) when the unlock was activated.
- Lines 80 to 87 are used to set `LKon` and `ULKon` to `FALSE` when the term of the current flashing function has expired.

3.3.6 Property 4

Property 4 of the *Flasher Manager* concerns the behavior of the *Flasher Manager* for an infinite time period. Practically, we can only check a bounded version of this property: we consider that the property is violated when the lights remain on for `d` consecutive time periods. The *C* program under verification for this bounded version of property 4 is displayed in Fig. 12. We thus introduce a loop bounded by `d` (line 9) that counts the number of times when the outputs of the *Flasher Manager* have consecutively been true (lines 26 to 39). After the loop, if these counters are equal to `d` (line 43), then the property is violated in the sense that the outputs have remained true during the whole time period that was considered. The value of the bound `d` is set as large as possible as shown in Sect. 4; its maximal value is mainly determined by the capabilities of the tools.

4 Experimental results

This section presents our experiments with constraint-based BMC for checking the properties of the *Flasher Manager* (see Sect. 3.2 and 3.3). For each property, we compare the results obtained with *DPVS* [11] to those obtained with *CBMC* [8], a state-of-the-art bounded model checker. Moreover, we also investigated the contribution of a Satisfiability Modulo Theories (SMT) solver to the resolution of the constraint problems issued from our experiments. Thus, each experiment was ran with the default solver of each tool and with the z3 SMT solver [23] as an alternative solver.

```
void prop3a(int d){
  init();
  _Bool doorsLocked=TRUE; // doors are closed
  // state variables for lock function
  _Bool oldLK=FALSE; // old value of LK
  _Bool LKon=FALSE; // no active function related to lock
  _Bool LK_on_lkd=TRUE; // lock when door is closed
  int countLK=-1; // number of cycles since LKon is true
  // state variables for unlock function
  _Bool oldULK=FALSE; // old value of LK
  _Bool ULKon=FALSE; // no active function related
                     // to unlock
  _Bool ULK_on_lkd=TRUE; // unlock when door is closed
  int countULK=-1; // number of cycles since ULKon
                   // is true
  for (int i=0;i<d;i++) {
    // input values which exclude L=R=1
    F=nondet_in(); W=nondet_in(); L=nondet_in();
    if (L) R=FALSE; else R=nondet_in();
    // input values of LK and ULK which exclude
    // combination of lock/unlock functions
    if (!LKon && !ULKon) {
      oldLK=LK; LK=nondet_in(); oldULK=ULK;
      if (LK) ULK=FALSE; else ULK=nondet_in();
      //  initialization of state variables
      if (LK && !oldLK) {// rising edge of LK
        LKon=TRUE; countLK=-1;
        if (!doorsLocked) {
          LK_on_lkd=FALSE; doorsLocked=TRUE;
        }
        else LK_on_lkd=TRUE;
      }
      if (ULK && !oldULK) {// rising edge of ULK
        ULKon=TRUE; countULK=-1;
        if (!doorsLocked) ULK_on_lkd=FALSE;
        else {
          ULK_on_lkd=TRUE; doorsLocked=FALSE;
        }
      }
    }
    // counters update
    if (LKon) countLK++;
    if (ULKon) countULK++;
    // call to f1
    f1();
```

Fig. 10. Function under verification for Property 3a: lock and unlock functions

```
    if (!F) {
      if (!LKon && !ULKon )
        // neither lock nor unlock is active
        assert(outL==FALSE && outR==FALSE);
      else {
        if (LKon) {
          // should be 1111111110000000000
          if (!LK_on_lkd && countLK>=2){
            if (countLK<=11)
               assert(outL==TRUE && outR==TRUE);
            else assert(outL==FALSE && outR==FALSE);
          }
          // should be 101010... during 60 cycles
          if (LK_on_lkd && countLK>=1){
            if (countLK%2!=0)
               assert(outL==TRUE && outR==TRUE);
            else assert(outL==FALSE && outR==FALSE);
          }
        }
        else { // ULKon is true
          // nothing shall happen
          if (!ULK_on_lkd && countULK>=1)
            assert(outL==FALSE && outR==FALSE)
          // should be 010101... during 20 cycles
          if (ULK_on_lkd && countULK>=1){
            if (countULK%2==0)
               assert(outL==TRUE && outR==TRUE);
            else assert(outL==FALSE && outR==FALSE);
          }
        }
      }
    }
    // to update LKon and ULKon when the flashing
    // period is finished
    if (LKon) {
      if (!LK_on_lkd && countLK==22) LKon=FALSE;
      if (LK_on_lkd && countLK==60) LKon=FALSE;
    }
    if (ULKon) {
      if (!ULK_on_lkd && countULK==2) ULKon=FALSE;
      if (ULK_on_lkd && countULK==20) ULKon=FALSE;
    }
  }
}
```

Fig. 11. Function under verification for Property 3a: lock and unlock functions (cont.)

First, we outline in Sect. 4.1 the strategies and tools involved in the experiments, then we detail the results, property by property, in Sect. 4.2.

4.1 Strategies and tools

DPVS is a strategy that we devised for constraint-based bounded model checking. *DPVS* was presented in details in Sect. 2.4; here, we only give an insight into the implementation. We also briefly recall the main features of *CBMC* and z3.

4.1.1 *DPVS*

DPVS is implemented in Comet [22]: 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

```
void prop4(int d) {
  // number of time where the left  light  has been
  // consecutively  true
  int countL = 0;
  // number of time where the right  light  has been
  // consecutively  true
  int countR = 0;
  // consider  d units  of time
  for(int i=0;i<d;i++) {
    // non-deterministic values  of the inputs
    L=nondet_in();
    if (L)
      R=FALSE;
    else
      R=nondet_in();
    W=nondet_in();
    LK=nondet_in();
    if (LK)
      ULK=FALSE;
    else
      ULK=nondet_in();
    F=nondet_in();
    // call  to f1() to simulate one pass through
    // the module
    f1();
    if (outL)
    // the  left  light  has been consecutively  true
    // one more time
      countL++;
    else
    // the  left  light  has not been consecutively  true
      countL=0;
    if (outR)
     // the right  light  has been consecutively  true
     // one more time
      countR++;
    else
     // the right  light  has not been consecutively  true
      countR=0;
   }
   // if  countL and countR are less than d,
   // then the  lights  did not remain lit
   assert (countL<d && countR<d);
}
```

Fig. 12. Function under verification for Property 4

stochastic combinatorial optimization with a language for modeling and searching[12].

Our prototype works from an `XML` representation of imperative programs. This allows our tool not to be tied to a single programming language. Simple translators from actual programming languages into the `XML` format are available, e.g., for *C* and Java. The current prototype imposes many restrictions on the *C* programs. Especially, input data are restricted to Booleans, integers and arrays of these primitive types. Run-time errors – e.g., division by zero– are not checked for. However, the prototype has all the *C* language features required for the *Flasher Manager* properties. In particular, function calls are inlined.

Comet has a default finite domain constraint solver which we used in the experiments; this default solver is denoted by CP in the results. However, Comet also allows to call other solvers in the form of external libraries. Thus, we developed an interface to call z3 instead of Comet's default solver.

4.1.2 *CBMC*

CBMC [8] is a state-of-the-art bounded model checker for ANSI-*C* and *C++* programs. It allows the verification of array bounds (buffer overflows), pointer safety, exceptions, and user-specified assertions. *CBMC* builds a propositional formula whose models correspond to execution paths of bounded length k violating some property of a program. This formula is then checked for satisfiability using a SAT solver. If the formula is satisfiable, the given property does not hold; otherwise, the property is guaranteed to hold up to k steps.

CBMC also offers the option to use z3 instead of the embedded SAT solver, but this is yet an experimental feature. For the experiments, we used *CBMC* version 3.6 since this was the latest version to correctly work with z3.

CBMC was called without any specific option except for specifying the unfolding bound with `--unwind` and the function to check with `--function`.

4.1.3 z3

A fundamental issue faced by model checkers is the state space explosion of the model. SMT-based model checking has been proposed as a generalization of SAT-based model checking to address this challenge. The idea is that parts of the model could be more easily expressed and more efficiently solved in more general, but still decidable, theories than propositional logic. SMT solvers integrate dedicated solvers for the different theories and share some of the motivations of constraint programming. SMT-based model checking has already been used with success in [2,12].

For the experiments, we used the SMT [23] solver z3 version 2.19, which supports:

- equality over free function and predicate symbols;
- real and integer arithmetic (with limited support for non-linear arithmetic);
- quantifiers;
- bit-vectors;
- arrays;
- tuple types and algebraic data-types.

[12] See `http://dynadec.com`

4.2 Results

We report in the following sections on the time spent by the tools to check the *Flasher Manager* properties. We give both *presolving* and *search* times when relevant.

- For *DPVS*, presolving time is used for building the simplified unfolded control flow graph from the `XML` representation of the program of a property.
  For *CBMC*, presolving time is the time spent to translate the *C* program into a Boolean formula.
  Presolving time also includes applying several simplification techniques (e.g., cone of influence, constant propagation), which may differ according to the tool.
- Search time is the time actually spent for searching for a solution, that is total time of the experiment minus presolving time. For *CBMC*, it corresponds to the time spent in the solver (SAT or SMT), whereas for *DPVS* it covers the time needed to dynamically explore the control flow graph and to solve the constraint systems.

A time limit of 10 minutes was allowed for each experiment, beyond which we stopped the experiment and reported a time-out (T.O.).

All the properties of the experiments are expressed as safety properties. In bounded model checking, the greater is the number of unfoldings the more confident one can be that the property holds. For the experiments, we started with a bound of 10 unfoldings which we increased next until we reached the given time limit.

All benchmarks were run on a 64-bit Linux quad-core Intel Xeon (3.16 GHz) platform with 16 GB of RAM. However, tools were run on a single core and did not take advantage of the three supplementary cores. Moreover, to our knowledge, *CBMC* 3.6 only exists in 32-bit version.

4.2.1 Property 1

Property 1 states that the warning function should have priority over the other flashing functions. This property does not hold: the lock and unlock flashing functions have priority over the warning function. All the tools found a counterexample with our starting bound of 10 unfoldings. Actually, 3 unfoldings are enough for the shortest counterexample. The results are shown in Table 1. z3 subscripts denote versions of the tools that use z3 instead of the native solvers of the tools.

CBMC performs very well on this property where a counterexample can be found with very few unfoldings. *DPVS* behaves also well: the presolving simplifications require more time but the search process is very fast. Note that the implementation of the presolving in *DPVS* has not been optimized at all.

On this property, the search time is much more important when z3 is used instead of the native solvers of the different systems.

Table 1. Presolving, search, and total times in seconds for checking Property 1 with 10 unfoldings

Tool	Presolving	Search	Total
CBMC	0.89	0.23	1.12
$CBMC_{z3}$	0.85	2.7	3.55
DPVS	3.89	0.08	3.97
$DPVS_{z3}$		0.34	4.23

Table 2. Total time in seconds for checking Property 3b (without lock and unlock functions) when varying the number of unfoldings

Unfoldings	*CBMC*	$CBMC_{z3}$	*DPVS*	$DPVS_{z3}$
10	0.56	0.7	3.04	3.44
50	6.56	7.11	16.59	21.02
100	44.71	53.72	37.47	55.92
200	202.18	288.76	94.8	155.33
300	510.91	T.O.	152.88	307.65
400	T.O.	T.O.	248.31	576.18

4.2.2 Property 2

This property is much more complex than Property 1: it involves nested loops and numerous conditional statements with combinations of disjunctions and conjunctions. None of the used tools could check the property within the 10-minute time limit, even for 10 unfoldings. *CBMC* did not finish building the Boolean formula. *DPVS* could build and simplify the control flow graph in around 40 s, but it reached the time limit during search. This property is not easy to reformulate in a more tractable form without losing its semantics. It shows that a relatively simple application as the *Flasher Manager* is still challenging for modern bounded model checking tools.

4.2.3 Property 3

The unrestricted version of Property 3, Property 3a, could not be checked within the 10-minute time limit. At least 120 unfoldings are required to include one sequence of all the lock and unlock related functions. However, for 100 unfoldings, *CBMC* fails to build the Boolean formula within the time limit. Our tool could build and simplify within the time limit the control flow graph for 100 unfoldings but *DPVS* failed to finish the graph exploration.

There are less combinations on the inputs for Property 3b since lock and unlock inputs are set to false. Figure 13 shows that *DPVS* and *CBMC* could check the property up to 300 unfoldings within the time limit. On small number of unfoldings, *CBMC* performs better than *DPVS*, but *DPVS* scales better than *CBMC*: at around 100 unfoldings and above, *DPVS* is faster. *DPVS* could even check the property for 400 unfoldings.

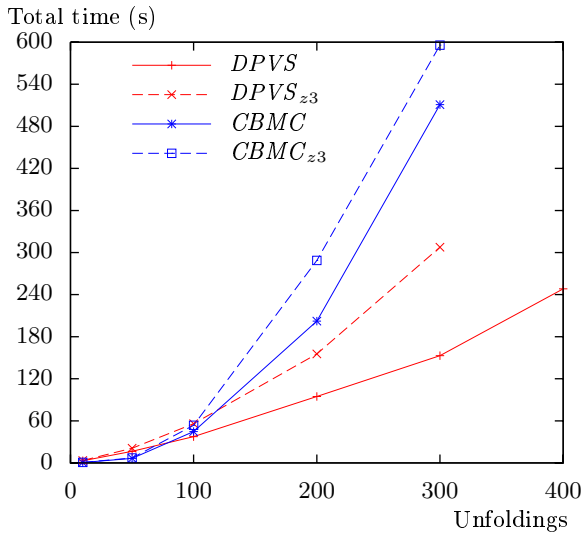


Fig. 13. Total time in seconds for checking Property 3b (without lock and unlock functions) when varying the number of unfoldings

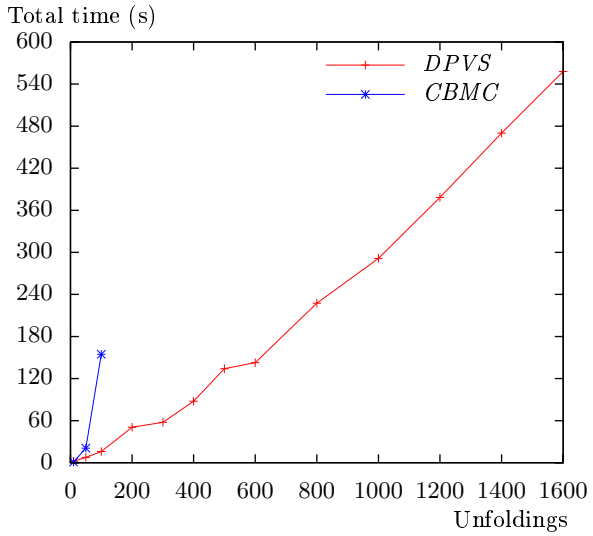


Fig. 14. Total time in seconds for checking Property 4 when varying the number of unfoldings

As with Property 1, the use of z3 increases the search time by a factor ranging from 2 to 4.

Neither *DPVS* nor *CBMC* found any counterexample within the unfolding bounds. This means that *DPVS* had to explore all executable paths relevant to this property.

4.2.4 Property 4

Property 4 is false if there are combinations of inputs so that one of the light remains always lit. However, in bounded model checking we cannot show that a light is actually *always* lit. We can only increase as much as possible the unfolding bound and show that on the corresponding time duration a light remains lit. Figure 14 reports the CPU time required to find a counterexample

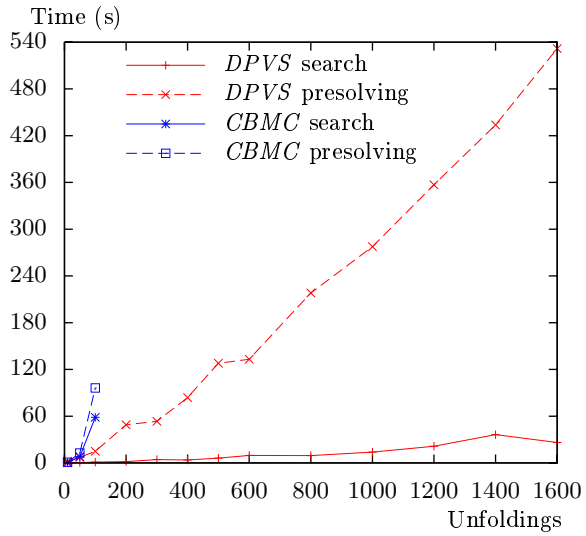


Fig. 15. Presolving and search times in seconds for checking Property 4 when varying the number of unfoldings

to Property 4 when the number of unfoldings increases. *DPVS* outperforms *CBMC* for this property. *DPVS* allows to reach up to 1600 unfoldings whereas *CBMC* fails at 200 unfoldings within the 10-minute time limit. In typical settings, 1600 cycles of the *Flasher Manager* control loop are more than 13 minutes of working.

Presolving and search times for *CBMC* and *DPVS* are given in Fig. 15. *DPVS* search time is very low compared to presolving time. In bounded model checking, the bound is usually progressively increased. So, in an optimized system the total checking time could benefit from an incremental building of the control flow graph from one unfolding bound to another, instead of rebuilding the graph from scratch for each bound as done here.

5 Discussion and related work

In this section, we first analyze the advantages and drawbacks of the different search strategies, then we discuss related work.

5.1 DPVS & CPBPV

We designed *DPVS* because *CPBPV* was not able to prove or disprove any property of the *Flasher Manager* application at a reasonable depth.

We also tested *DPVS* on other well known examples: the *Tritype* program for triangle classification [10], the *TCAS* program from avionics [19], and the *Binary search* that determines whether a value v is present in a sorted array t. *DPVS* and *CPBPV* have similar performances on the two first examples, and compare favorably with *CBMC*. The main impact on performances for both *DPVS* and *CPBPV* is the way we combine the solvers (i.e., CP and LP solvers).

Table 3. Presolving, search, and total times in seconds for checking Property 4 when varying the number of unfoldings

Unfoldings	*CBMC*			*DPVS*		
	Presolving	Search	Total	Presolving	Search	Total
10	0.84	0.15	0.99	2.02	0.09	2.11
50	12.72	8.08	20.8	7.19	0.5	7.69
100	96.21	58.52	154.73	14.9	1.11	16.01
200	T.O.	T.O.	T.O.	48.99	1.7	50.69
300	T.O.	T.O.	T.O.	53.42	4.27	57.68
400	T.O.	T.O.	T.O.	83.81	3.83	87.64
500	T.O.	T.O.	T.O.	127.97	6.13	134.09
600	T.O.	T.O.	T.O.	132.97	9.62	142.59
800	T.O.	T.O.	T.O.	218.15	9.35	227.5
1000	T.O.	T.O.	T.O.	277.47	13.91	291.39
1200	T.O.	T.O.	T.O.	356.93	21.4	378.33
1400	T.O.	T.O.	T.O.	433.82	36.24	470.05
1600	T.O.	T.O.	T.O.	531.82	26.2	558.02

Table 4. Time in seconds for checking the Binary search benchmark when varying the array length (with 16-bit integers)

Array length	CBMC	DPVS	*CPBPV*
4	5.732	0.529	0.107
8	110.081	35.074	0.298
16	T.O.	T.O.	1.149
32	T.O.	T.O.	5.357
64	T.O.	T.O.	27.714
128	T.O.	T.O.	153.646

On the contrary, *DPVS* and *CPBPV* have very different performances on the *Binary search* example. The characteristic of this example is that it is correct, which means that all executable paths must be explored, and above all, it has a very strong pre-condition. Thus a top-down strategy like *CPBPV* is well adapted to take advantage of this pre-condition for cutting unfeasible paths (i.e., paths where the array is not sorted).

Table 4 reports the results of the experiments on a correct version of the *Binary search* program. CBMC and *DPVS* do not handle this benchmark. CBMC wastes a lot of time in building and exploring the whole formula. *DPVS* strategy 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 pre-condition. This pre-condition combined with the decisions taken along a path have a strong impact on feasibility of the next conditions and help to prune infeasible paths.

This benchmark clearly shows that neither *DPVS* nor *CBMC* work well on all problems. Of course, this is due to the combinatorial complexity behind the used algorithms. However, we can point out that *DPVS* and *CBMC* are much more efficient than *CPBPV* for (partially) proving properties on real applications without a complete formal specification.

5.2 Related work

Standard bounded model checkers transform a program and a property into a Big Boolean formula and use SAT solvers to prove that the property holds or to find a counterexample [16]. SMT solvers are now used in most of the state-of-the-art BMC tools to directly work on high-level formula (see [2,17,13] and *CBMC*). Many improvements have been studied for high-level BMC, such as the one proposed in [17], in particular during the unfolding step and to reuse previously learned 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., [20,21,26,1]) and provides a nice implementation tool extending symbolic execution techniques [7]. Gotlieb *et al.* showed how to represent imperative programs as constraint logic programs: InKa [20] 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 [15]. 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.

Euclide [18] is also a successor of InKa. It has three main functions: structural test data generation, counterexample generation and partial program proving for critical *C* programs. Euclide builds the constraints in an incremental way and combines standard constraint programming techniques and specific techniques to handle floating point numbers and linear constraints.

6 Conclusion

In this paper, we have introduced *DPVS*, a dynamic constraint based strategy for bounded model checking. 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 an unoptimized prototype whereas CBMC is a state-of-the-art solver.

The non-sequential strategy of *DPVS* is very well adapted for problems with a strong post-condition. In contrast, the top-down strategy of *CPBPV* is much more efficient for problems with a strong pre-condition, like the Binary search benchmark. Clearly, it would be worth investigating new strategies combining the capabilities of *DPVS* and *CPBPV*.

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 [7], to be able to evaluate the proposed approach on a larger class of programs.

We are also working on a solver more adapted to software verification problems, for instance a solver that handles disequalities in a more efficient way. 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 $x \neq y$. If the domains are large, consistency checks with CP may be very costly; if we want to use LP, we have to create two choice points: $x < y$ and $x > y$.

In this paper, we have also provided a new industrial case study. We have proposed models of the different properties to prove[13] and evaluated different tools and strategies. Although this application seems quite simple in regard to other applications like ABS controllers, some properties could neither be proven nor disproven by the tools we evaluated. So, it is still a challenge for state-of-the-art bounded model checkers.

References

1. Albert, E., Gómez-Zamalloa, M., Puebla, G.: Test Data Generation of Bytecode by CLP Partial Evaluation. In: LOPSTR 2008(Logic-Based Program Synthesis and Transformation), Revised Selected Papers, *LNCS*, vol. 5438, pp. 4–23. Springer (2008)
2. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. International Journal on Software Tools for Technology Transfer **11**(1), 69–83 (2009)
3. Ball, T., Levin, V., Rajami, S.K.: A Decade of Software Model Checking with SLAM. CACM **54**(7), 68–76 (2011)
4. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. Information Processing Letters **93**(6), 281–288 (2005)
5. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: TACAS, pp. 193–207. Springer (1999)
6. Bochot, T., Virelizier, P., Waeselynck, H., Wiels, V.: Model checking flight control systems: The Airbus experience. In: ICSE 2009 (31st International Conference on Software Engineering),Companion Volume, pp. 18–27. IEEE (2009)
7. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Softw. Test., Verif. Reliab. **16**(2), 97–121 (2006)
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS, *LNCS*, vol. 2988, pp. 168–176 (2004)
9. Collavizza, H., Rueher, M., Hentenryck, P.V.: CPBPV: A Constraint-Programming Framework for Bounded Program Verification. In: CP 2008 (14th International Conference on Principles and Practice of Constraint Programming), *LNCS*, vol. 5202, pp. 327–341. Springer (2008)
10. Collavizza, H., Rueher, M., Hentenryck, P.V.: A constraint-programming framework for bounded program verification. Constraints Journal **15**(2), 238–264 (2010)
11. Collavizza, H., Vinh, N.L., Rueher, M., Devulder, S., Gueguen, T.: A dynamic constraint-based bmc strategy for generating counterexamples. In: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011, pp. 1633–1638. ACM (2011)
12. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09), pp. 137–148. IEEE Computer Society (2009)
13. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. ASE **0**, 137–148 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/ASE.2009.63
14. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
15. Denmat, T., Gotlieb, A., Ducassé, M.: Improving constraint-based testing with dynamic linear relaxations. In: Proc. of ISSRE, The 18th IEEE International Symposium on Software, pp. 181–190. IEEE Computer Society (2006)
16. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. on CAD of Integrated Circuits and Systems **27**(7), 1165–1178 (2008)
17. Ganai, M.K., Gupta, A.: Accelerating high-level bounded model checking. In: International Conference on Computer-Aided Design (ICCAD'06), pp. 794–801 (2006)
18. Gotlieb, A.: Euclide: A Constraint-Based Testing Framework for Critical C Programs. In: ICST 2009, Second International Conference on Software Testing Verification

[13] All these programs, as well as the source code of the *Flasher Manager* application, are publicly available at `http://users.polytech.unice.fr/~rueher/Benchs/FM`.

and Validation, 1-4 April 2009, Denver, Colorado, USA, pp. 151–160. IEEE Computer Society (2009)
19. Gotlieb, A.: TCAS software verification using constraint programming. The Knowledge Engineering Review, (Accepted for publication) (2010)
20. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: ISSTA,International Symposium on Software Testing and Analysis, pp. 53–62 (1998)
21. Jackson, D., Vazir, M.: Finding bugs with a constraint solver. In: ISSTA, International Symposium on Software Testing and Analysis, pp. 14–25. ACM Press (2000)
22. Michel, L., Hentenryck, P.V.: The comet programming language and system. In: P. van Beek (ed.) Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05), *LNCS*, vol. 3709, pp. 881–881. Springer (2005)
23. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: C.R. Ramakrishnan, J. Rehof (eds.) Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), *LNCS*, vol. 4963, pp. 337–340. Springer (2008)
24. Régin, J.C.: A filtering algorithm for constraints of difference in csps. In: AAAI, pp. 362–367 (1994)
25. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier (2006)
26. Sy, N.T., Deville, Y.: Automatic test data generation for programs with integer and float variables. In: ASE (16th IEEE International Conference on Automated Software Engineering), pp. 13–21. IEEE Computer Society (2001)