Preamble Computation in Automated Test Case Generation using Constraint Logic Programming

Séverine Colin, Bruno Legeard, and Fabien Peureux

Laboratoire d'Informatique (LIFC)
Université de Franche-Comté - CNRS - INRIA
16, route de Gray - 25030 Besançon, France
Tel.: (33) 381 666 515
Fax.: (33) 381 666 450
e-mail: {colin,legeard,peureux}@lifc.univ-fcomte.fr
http://lifc.univ-fcomte.fr/∼bztt

Abstract. BZ-TESTING-TOOLS (BZ-TT) is a tool-set for automated model-based test case generation from B abstract machines and Z specifications. BZ-TT uses boundary testing as well as cause-effect testing on the basis of the formal model. It has been used and validated on several industrial case studies in the domain of critical software: in particular for smart card applications and automotive embedded systems. The main idea of BZ-TT is to compute a boundary goal for each effect of the operations of the model and then to compute a preamble sequence of operations to place the system under test in a such a state that satisfies the goal.
In this paper, the preamble computation search strategies used in BZ-TT are presented. More precisely, two algorithms based respectively on forward chaining and backward chaining are compared. These algorithms all use a customized set constraint solver, which is able to animate the formal model. These algorithms differ, however, in their capacity to efficiently reach the boundary goals. The results of applying the tools to an industrial windscreen wiper controller application are presented.
Keywords: model-based testing, preamble computation, set constraint solving, B notation, forward chaining, backward chaining.

1 Introduction

Set-oriented formal specification notations, such as Z [1], VDM [2] and B [3] have been recognized since the beginning of the 1990s as a suitable basis for model-based testing (see for example [4, 5, 6, 7, 8]). In this framework, the formal model is used as an input for test case generation in order to validate an implementation under test. Test cases are generated on the basis of a partition analysis of the formal model. For example, Dick and Faivre [4] derive a partition based on the disjunctive normal form of the invariant and the operations. A key issue, however, to fully automating the test generation process is the sequencing of operation invocations. Some authors [4, 5, 6] have used partition analysis to build an underlying Finite State Automaton (FSA) and have then computed test sequences as paths in this FSA. Unfortunately, this method is not easily automated, because it is difficult to choose an appropriate abstraction of the state space to generate the FSA. Other authors [9] have used hybrid specifications, such as Statecharts with Z. This approach allows the production of an extended finite state machine to model operation sequences. In this case, it is impossible to use set-oriented formal specification notations as they are.

In [10], a novel approach to automate test generation from B abstract machines or Z specifications, based on a customized set constraint solver [11], was introduced. This method, called BZ-TESTING-TOOLS (BZ-TT), also uses partition analysis, and computes *effect predicates* as partitions for each operation of the model. The basic idea of the BZ-TT method is to cover all effect predicates at boundary state variable values with boundary input values. Some effect predicates might not apply to the model's initial state. If so, it is necessary to carry out a sequence of preliminary operations (generally referred to as a preamble computation) to place the model in a state where such an effect predicate can be invoked. Determining appropriate preambles is widely recognised as a challenging task. The method described in this paper avoids the construction of a complete FSA from the formal model and provides also a strong boundary-value testing orientation. This test generation method is embedded in the BZ-TT environment [12], which has been used on several industrial applications in two domains: smart card software (the GSM 11-11 standard [13, 14], the Java Card transaction mechanism [15]), and automobile embedded control-command software (a windscreen wiper controller and a heating and ventilation controller [16]). These industrial applications showed that the efficiency of the preamble computation procedure is of major importance to the scalability of the BZ-TT approach.

This paper focuses on how to sequence operation invocations in model-based test generation with set-oriented formal specification notations. More precisely, it is a study of how various algorithms traverse the reachability graph which underlies the formal model in order to reach particular target states, i.e. the boundary states. Two main search procedures have been compared on the basis of an industrial case-study: forward chaining and backward chaining. This paper shows that backward chaining is better suited for reaching some boundary states than forward chaining. This study extends the work by Pretschner [17], which uses classical search strategies to explore the space of reachable states in order to generate tests.

This paper is structured as follows. Section 2 introduces the BZ-TT test generation method in greater detail. Section 3 defines the preamble computation. Sections 4 and 5 respectively present the two preamble computation algorithms: forward and backward chaining. Section 6 gives the results based on the windscreen wiper controller application. Finally, Section 7 presents conclusions.

2 Overview of the BZ-TT Test Generation Method

The implementation of the system under test is usually a state machine with hidden states. This state machine is specified by a B abstract machine or Z schema, which has a state space (consisting of several state variables) and several operations that are specified via pre-conditions and post-conditions. The operations are characterized by a name and possible input and output parameters. If an operation modifies the system state it is called an *update operation*; if not, it is called an *observation operation*.

The B specifications, inputs of our test generation method, form abstract models of the requirements. That means that only the machine level is used (no refinement or implementation level). Moreover, abstract sets must be replaced by finite enumerated sets for the purposes of test generation. This enables our CLP-based tools to perform much stronger reasoning about the specification, and this is usually necessary for the test-generation process to be tractable. These two restrictions were never a problem for the industrial case-studies.

Before presenting the main stages of the BZ-TT test generation method, the next subsection introduces the B abstract machine of the simplified windscreen wiper controller example, which will be used throughout this paper as a working example.

Then, the set constraint solver, called CLPS-B [11, 18] is briefly introduced. This solver constitutes the basis of our test generation method presented in the last part of this section.

2.1 Simplified Windscreen Wiper Controller Example

The *WIPER_SPEED* B abstract machine, shown in Figure 1, is a highly simplified version of a windscreen wiper speed controller formal model.

The user can turn the windscreen wiper *on* or *off* by using the control lever. The windscreen wiper has three states: stopped, intermittent or continuous. If the control lever is off, the wiper state is stopped; otherwise it is intermittent or continuous. These two last states directly depend on the car speed. In fact, the wiper is always continuous, except when the car first exceeds 10 km/h and then drops to under 10 km/h (in this case the wiper becomes intermittent). Afterward, the wiper can be continuous if the car again exceeds 10 km/h. To know if the car has exceeded 10 km/h when the control lever was on, the variable *flag* is introduced.

This abstract machine has two update operations (*speed* and *action_lever*) and one observation operation (*alert*). *Speed* and *action_lever* modify the car speed and the control lever value respectively. The *alert* operation is a very abstract operation that has been introduced to illustrate local minimum problems. It can be triggered when the wiper is intermittent and the car speed equals zero.

Finally, in the initial state, the car speed is 0, the control lever is off, the flag is false and the wiper is stopped.

2.2 Test Case Definition using CLPS-B

The behaviour of such a system can be described in terms of a sequence of operations – a trace – where the first is activated from the initial state of the system.

If the pre-condition of an operation is false, its post-condition cannot be guaranteed to hold after invocation. Any subsequent operations are of no interest since it is impossible to predict the eventual state of the machine. Thus, a test case can be defined as any legal trace, i.e. any trace where all pre-conditions are true. The submission of a legal trace is successful if all the output values returned by the concrete implementation during the trace are equivalent (through a function of abstraction) to the output values returned by the formal model of the implementation during the simulation of the same trace (or included in the set of possible values if the formal model is non-deterministic).

The BZ-TT method is fully supported by the BZ-Testing-Tools tool-set. This environment is a set of tools dedicated to animation and test case generation from B or Z formal specifications. It is based on a constraint solver, called CLPS-B, able to evaluate and perform execution of B and Z formal models. By execution, we mean that the solver computes a so-called constrained state by applying the pre- and post-condition of operations. A constrained state is a constraint store where each state variable, input variable and output variable supports constraints. The solver can also verify the invariant properties on any constrained state. Because of the set oriented feature of B and Z notations, CLPS-B is mainly a set constraints solver. Its constraints domain is hereditary finite sets with nested pairs in order to reason about relations and mappings. To compute the post-condition the constraint solving makes use set interval propagation [19] and specific propagation

```
MACHINE
    WIPER_SPEED
SETS
    $CONTROL\_LEVER = \{on, off\}$;
    $BOOLEAN = \{false, true\}$
    $WIPER\_STATE = \{stopped,$
            $intermittent, continuous\}$
DEFINITIONS
    $SPEED == 0..100$
VARIABLES
    lever,
    car_speed,
    wiper,
    flag
INVARIANT
    $lever \in CONTROL\_LEVER \land$
    $car\_speed \in SPEED \land$
    $wiper \in WIPER\_STATE \land$
    $flag \in BOOLEAN$
INITIALISATION
    $lever := off \parallel$
    $car\_speed := 0 \parallel$
    $wiper := stopped \parallel$
    $flag := false$

OPERATIONS
    alert =
        PRE
            $wiper = intermittent \land$
            $car\_speed = 0$
        THEN
            skip
        END;

    speed(sp) =
        PRE
            $sp \in SPEED$
        THEN
            $car\_speed := sp \parallel$
            IF $(sp > 10 \land lever = on)$
            THEN    $wiper := continuous \parallel$
                    $flag := true$
            ELSE
                IF $(sp \leq 10 \land flag = true)$
                THEN    $wiper := intermittent$
                END
            END
        END;

    action_lever(action) =
        PRE
            $action \in CONTROL\_LEVER \land$
            $action \neq lever$
        THEN
            $lever := action \parallel$
            IF $action = off$
            THEN    $wiper := stopped \parallel$
                    $flag := false$
            ELSE
                $wiper := continuous \parallel$
                IF $car\_speed > 10$
                THEN    $flag := true$
                END
            END
        END
    END
```

Fig. 1. Simplified Windscreen Wiper Speed Controller B Abstract Machine

rules. For basic arithmetic operations and constraints solving on set cardinality, there is a cooperation between set constraints propagation and the integer finite domains solver CLP(FD) of Sicstus-Prolog [20]. CLPS-B uses partial consistency techniques [21] during constraint propagation; the completeness is ensured by classical labelling using AC3 algorithm [22] thanks to the finiteness of abstract model data structure. The correctness is ensured by the translation rules from abstract model into the system of constraints.

The BZ-TT method tests operations when the system is in its various *boundary states*. A boundary state is a state where at least one variable has a value at an extremum – minimum or maximum – of its sub-domains. The method works by computing a set of *boundary goals* from the Disjunctive Normal Form (DNF) of the specification. The CLPS-B solver is also used to compute the boundary goals, and to generate, by simulating the execution of operations, the test cases. A major feature of the CLPS-B solver is that it avoids the construction of a complete finite state automaton (FSA) for the system: test cases are generated by traversal of the state space defined by the specification.

The computation of the DNF form (via *effect predicates*) is presented and illustrated with the windscreen wiper speed controller example in the next subsection. Subsection 2.4 presents the computation of the boundary goals.

2.3 Effect Predicate Computation

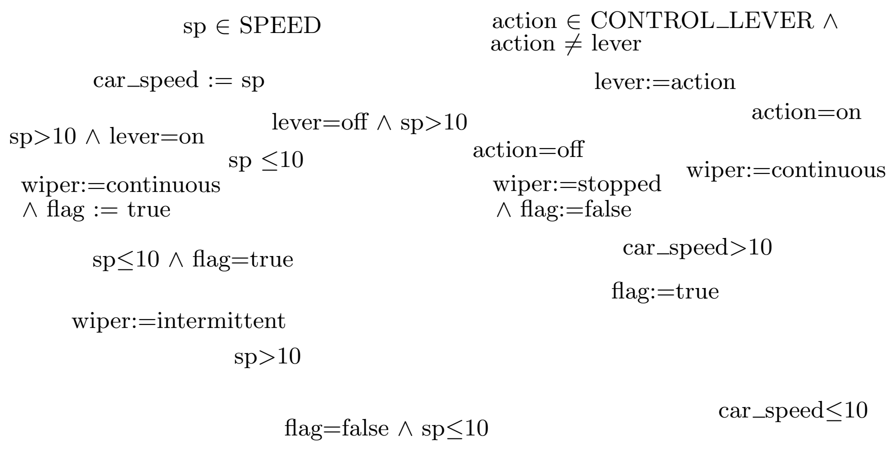
The BZ-TT environment supports both B abstract machines and Z specifications by translating them into a common notation, called BZP [12]. This notation is the input of the CLPS-B solver.

The BZP format is an equivalent before-after predicate form of the input formal model. For Z, the translation is obvious, using the prime variables. For B, the translation scheme from B generalized substitution to before-after predicates is precisely defined in the B-Book [3]. For example, the operation PRE P THEN S END (where P and S respectively define the preconditions and the substitutions of the operation) is interpreted as:

$$Context \wedge Inv \wedge P \wedge \exists\, x \cdot prd_x(S)$$

where the context (*Context*) and invariant (*Inv*) are predicates given in the abstract machine and *prd* is the transformer function in the before-after predicates (the substitution S is always feasible with termination) and x is the set of all the state variables updated by the substitution.

The BZP format makes it possible to partition each operation of the specification by determining the different effects of the operations using before-after predicates. These predicates are called *effect predicates.* Basically, effect predicates are computed by traversing the control flow-graph of each model operation (see [23] for more details). The *or* operators give rise to two edges into the control flow graph. The $A \vee B$ formula gives a first edge with A and the second with $\neg A \wedge B$.



(a) speed operation (b) action lever operation

Fig. 2. BZP flow-graph of the simplified windscreen wiper speed controller example

In the simplified windscreen wiper speed controller B model, the BZP format reveals one effect predicate for the *alert* operation, four for the *speed* operation

(Figure 2(a)), and three for the *action_lever* operation (Figure 2(b)). Effect predicates are separately given in Table 1. *Context* and *Inv* are not written but they must be considered for each effect predicate such as:

- *Context* is $CONTROL\_LEVER = \{on, off\} \land BOOLEAN = \{false, true\} \land SPEED == 0..100 \land WIPER\_STATE = \{stopped, intermittent, continuous\}$
- *Inv* is $lever \in CONTROL\_LEVER \land car\_speed \in SPEED \land wiper \in WIPER\_STATE \land flag \in BOOLEAN$

Operations	**N^o**	**Effect Predicates**
alert	E_1	wiper = intermittent ∧ car_speed = 0
speed	E_2	sp ∈ SPEED ∧ sp > 10 ∧ lever = on ∧ car_speed' = sp ∧ wiper' = continuous ∧ flag' = true
	E_3	sp ∈ SPEED ∧ sp ≤ 10 ∧ flag = true ∧ car_speed' = sp ∧ wiper' = intermittent
	E_4	sp ∈ SPEED ∧ sp ≤ 10 ∧ flag = false ∧ car_speed' = sp
	E_5	sp ∈ SPEED ∧ lever = off ∧ sp > 10 ∧ car_speed' = sp
action_lever	E_6	action ∈ CONTROL_LEVER ∧ action ≠ lever ∧ action = off ∧ lever' = action ∧ wiper' = stopped ∧ flag' = false
	E_7	action ∈ CONTROL_LEVER ∧ action ≠ lever ∧ action = on ∧ car_speed > 10 ∧ lever' = action ∧ wiper' = continuous ∧ flag' = true
	E_8	action ∈ CONTROL_LEVER ∧ action ≠ lever ∧ action = on ∧ car_speed ≤ 10 ∧ lever' = action ∧ wiper' = continuous

Table 1. Effect predicates of the windscreen wiper speed controller operations

Actually, effect predicates correspond to paths in the control flow graphs of the operations. To avoid generating tests from inconsistent effect predicates, the BZ-TT environment deletes every effect predicate E_i such that $Context \land Inv \land E_i$ is unsatisfiable. For example, in the speed operation, the predicate $sp \leq 10 \land sp > 10$ is unsatisfiable and is removed.

2.4 Boundary Goal Computation

Boundary goals are computed on the basis of the effect predicates by minimization and maximization using a suitable metric function depending on the boundary coverage criteria [23] chosen by the tester. This results in one or several minimum and maximum boundary goals for each effect predicate.

Given the invariant properties *Inv*, the context properties *Context*, the effect predicate E_i, the vector of variables V_i, which comprises all the free-state variables within E_i, and the optimization function f, the boundary goals are computed as follows:

$$BG_i^{min} = minimize(f(V_i), Inv \land Context \land E_i) \quad (1)$$
$$BG_i^{max} = maximize(f(V_i), Inv \land Context \land E_i) \quad (2)$$

The optimization function $f(V_i)$, where V_i is a vector of variables $v_1 \ldots v_m$, is defined as $g_1(v_1) + g_2(v_2) + \ldots + g_m(v_m)$, where each function g_i is chosen according to the type of the variable v_i.

From the effect predicates of the windscreen wiper speed controller operations, boundary goals BG^{min} and BG^{max} are computed with the optimization function $\sum_{v \in V_i} v$ if V_i is a vector of numeric variables, and the existential quantifier ($\exists$) if it is not (the variable is arbitrary assigned to one value of its domain). For example, the result of constraint solving using the effect predicate E_8 is a set of constraints on the *lever* and *car_speed* variables such that: $Inv \wedge Context \wedge E_i \equiv lever = off \wedge 0 \leq car_s peed \leq 10$. Applying minimization and maximization on these two state variables give the two following boundary goals :

– $lever = off \wedge car\_speed = 0$ (minimization)
– $lever = off \wedge car\_speed = 10$ (maximization)

All the boundary goal results are shown in Table 2.

Operations	**Effect Predicates**	**Boundary goals**	
alert	E_1	wiper = intermittent $\wedge$ car_speed = 0	BG_1
speed	E_2	lever = on	BG_2
	E_3	flag = true	BG_3
	E_4	flag = false	BG_4
	E_5	lever = off	BG_5
action_lever	E_6	lever = on	BG_2
	E_7	lever = off $\wedge$ car_speed = 11	BG_6
	E_7	lever = off $\wedge$ car_speed = 100	BG_7
	E_8	lever = off $\wedge$ car_speed = 0	BG_8
	E_8	lever = off $\wedge$ car_speed = 10	BG_9

Table 2. Boundary goals from the windscreen wiper speed controller example

The BZ-TT method consists in instantiating each boundary goal into a *boundary state* using preamble computation. A boundary state is a fully instantiated state obtained by the traversal - i.e. the preamble computation - of the state space according to a certain boundary goal.

Note that some effect predicates (and thus boundary goals) may be satisfiable, but still not reachable, because the states that satisfy $Inv \wedge Context \wedge E_i$ are not reachable by any sequence of operations. This can happen when the invariant is not the strongest possible invariant. The non-reachability of the effects cannot be checked locally, since it is a global property of the system. Also, the boundary goals may be not reachable when the corresponding effect predicate is. In this case, the preamble is computed to reach the complete effect predicate (not the boundary instantiation of it).

From this boundary state, all the possible behaviours of the specification must be tested. That is, the goal is to invoke each *update operation* with extremum values of the sub-domains of the input parameters called *boundary inputs* (Table 3). The process of boundary-value analysis for input variables is similar to that for state variables.

The computation of the boundary states, and more generally of the test cases, is defined in the next subsection.

Operations	Effect Predicates	Boundary inputs	
speed	E_2, E_5	sp = 11	BI_1
	E_2, E_5	sp = 100	BI_2
	E_3, E_4	sp = 0	BI_3
	E_3, E_4	sp = 10	BI_4
action_lever	E_6	action $\neq$ lever $\wedge$ action = off	BI_5
	E_7, E_8	action $\neq$ lever $\wedge$ action = on	BI_6

Table 3. Boundary inputs from the windscreen wiper speed controller example

2.5 Test Case Computation

The trace that constitutes the test case is divided into four sub-sequences[1] (see Figure 3):

Preamble: this takes the system from its initial state to a boundary state.
Body: this invokes one update operation with input boundary values.
Identification: this is a sequence of observation operations to strengthen the pass/fail verdict of the test case.
Postamble: this takes the system back to the boundary state or an initial state. This enables test cases to be concatenated for automated execution.

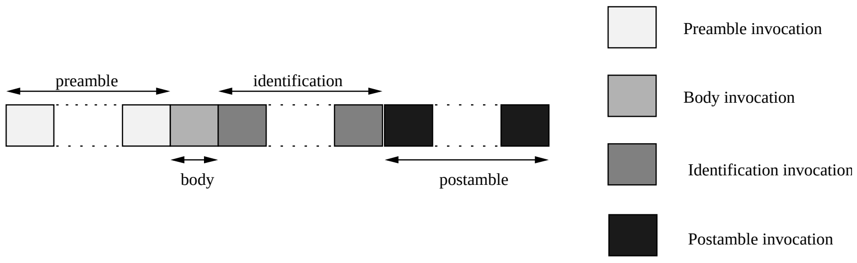


Fig. 3. Test case constitution

Update operations are used in the preamble, body and postamble and observation operations are used in the identification part.

The BZ-TT generation method is defined by the following algorithm, where $\{bound_1, bound_2, ..., bound_n\}$ and $\{op_1, op_2, ..., op_m\}$ respectively define the set of all boundary goals and the set of all the update operations of the specification:

```
for i=1 to n                % for each boundary state
    preamble($bound_i$);        % reach the boundary state
    for j=1 to m            % for each update operation
        body($op_j$);            % test $op_j$
        identification;     % observe the state
        postamble($bound_i$);   % return to the boundary state
    endfor
    postamble(init);        % return to the initial state
endfor
```

[1] The vocabulary follows the ISO9646 standard [24].

This algorithm computes test cases using boundary input values at body invocations. Figure 4 shows how the state space is traversed during the generation method. A set of one or more test cases, concatenated together, defines a test suite.

It should be noted that BZ-TT uses specific coverage criteria for complex data structures based on boundary analysis. These criteria allows the test engineers to control a possible test case explosion [23].

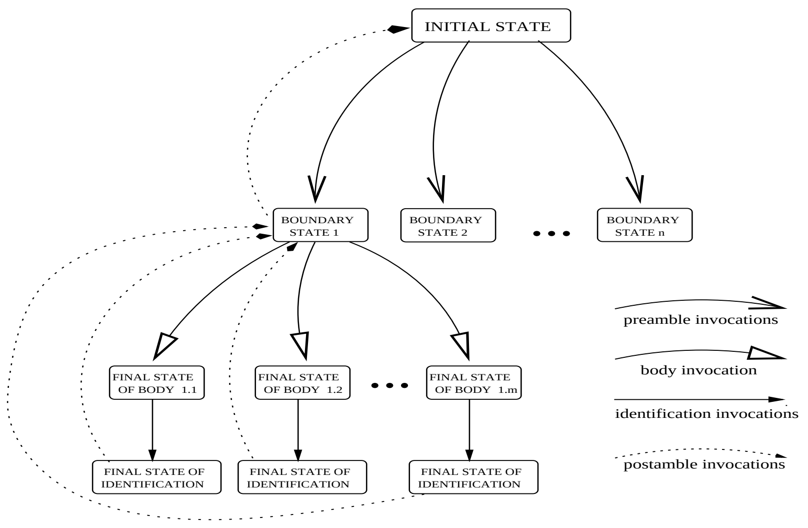


Fig. 4. Traversal of the state space during the test sequence generation.[2]

The user can choose to generate tests without an identification or without a postamble. If there is no postamble, we assume that the system is initialised between each tests. The oracle is computed from the state variable values and/or the identification.

It is not always possible to return to the boundary state or to the initial state. The postamble computation can fail to find a path. In these cases, the method assumes the possibility of a reset to the initial state followed by a preamble again. Therefore, the success of the preamble computation procedure is fundamental for the scalability of the BZ-TT method and tool-set; the generation of (possibly many) tests from a goal state depends upon first reaching that state via a successful preamble. This procedure using the CLPS-B solver is based on a symbolic execution of the formal model, that it, to compute the next constrained state after invoking a model operation and then to dynamically traverse the reachability graph underlying the formal model.

If we choose to generate tests without an identification and without a postamble with the simplified windscreen wiper example, we obtain 40 tests (the boundary goal BG_5 is not used because it is included in the boundary goal BG_6). There are 5 bodies for each preamble. For example, Table 4 shows the 5 tests generated from the boundary goal BG_3.

There are several search strategies available for computing the preamble. In the remainder of this paper the preamble will be defined and then two main

[2] The final state of the body and the final state of identification are generally the same, because observation operations may not affect the value of the state variables.

Operations	**States after the operation invocations**
	Preamble
action_lever(on)	lever = on ∧ car_speed = 0 ∧ wiper = continuous ∧ flag = false
speed(11)	lever = on ∧ car_speed = 11 ∧ wiper = continuous ∧ flag = true
	Body
action_lever(off)	lever = off ∧ car_speed = 11 ∧ wiper = stopped ∧ flag = false
speed(0)	lever = on ∧ car_speed = 0 ∧ wiper = intermittent ∧ flag = true
speed(10)	lever = on ∧ car_speed = 10 ∧ wiper = intermittent ∧ flag = true
speed(11)	lever = on ∧ car_speed = 11 ∧ wiper = continuous ∧ flag = true
speed(100)	lever = on ∧ car_speed = 100 ∧ wiper = continuous ∧ flag = true

Table 4. Generated tests from boundary goal BG_3 controller example

search strategies will be presented and compared : forward chaining and backward chaining.

3 Preamble Computation

The preamble computes traces in order to reach the states that satisfy boundary goals. The goal of the preamble is to compute one trace (the shortest trace is not required).

In BZ-TT, the preamble computation uses a best-first search [17] to find a trace between the initial state and a boundary state that satisfies a given boundary goal. The best-first search procedure is described in Figure 5.

```
compute_preamble(startstate) → status
begin
    frontier := {(startstate)};
    evaluatedVertices := {(startstate)};
    depthBoundReached := false;
    while frontier ≠ ∅ ∧ #evaluatedVertices ≤ maxEvaluatedVertices do
        best :∈ minCostTraces(frontier);
        others := frontier - {best};
        if isSolution(best) then
            return found;
        endif
        if length(best) ≤ maxDepth then
            depthBoundReached := true;
            frontier := others;
        else
            children := computeChildren(best);
            frontier := children ∪ others;
            evaluatedVertices := evaluatedVertices ∪ children
        endif
    endwhile
    if #evaluatedVertices > maxEvaluatedVertices then
        return unreachable_at_this_vertex_number;
    elsif frontier = ∅ ∧ depthBoundReached then
        return unreachable_at_the_depth
    else return unreachable
    endif
end
```

Fig. 5. Best-first search algorithm

The best-first search can be used both with the forward chaining procedure or the backward chaining procedure. With forward chaining the search starts at the initial state (*startstate*), whereas with backward chaining it starts at the boundary goal that has to be reached. From the root, a function *compute_children* is applied in order to compute the children of a vertex. The forward and backward chaining approaches have different *compute_children* functions. The respective functions are described Section 4 and 5.

The algorithm succeeds (function *is_solution*) when the current vertex satisfies:

– the boundary goal predicate in forward chaining,
– the initial state in backward chaining.

The algorithm is performed using a boundary depth and a boundary-valuated vertex number. The variable *frontier* represents all the vertices which have not yet been evaluated. The variable *evaluatedVertices* represents all the computed vertices. The variable *depthBoundReached* shows if the boundary depth has been reached during the search. The function *minCostTraces* returns the best traces using a cost function.

The algorithm can end within one of the following four states:

– *found*, a solution is found,
– *unreachable*, the boundary goal is unreachable,
– *unreachable_at_this_depth*, the boundary goal is unreachable at this depth,
– *unreachable_at_this_vertex_number*, the boundary goal is unreachable with the vertex number limitation.

In the next two sections, the two search procedures (forward and backward chaining) will be presented and evaluated on the simplified windscreen wiper controller example.

4 Forward Chaining

Preamble computation in forward chaining can be viewed as a traversal of the reachability graph, of which vertices represent the constrained states built during the simulation, and transitions an operation invocation.

Definition 1. *A constrained state is a pair $\theta_i(V, C_i)$ where V is the state variable set of the specification, and C_i is the set of the constraint variables.*

A consequence of this preamble computation is that the state variables which are not already assigned to a value by the boundary goal, are assigned a reachable value within their domain. Each boundary goal is also instantiated to one or more reachable boundary states by exploring the reachable states of the system, starting from the initial state. The CLPS-B solver simulates the execution of the system, recording the set of possible solutions after each operation.

Definition 2. *Let $activ_{EP}$: constrained_state $\rightarrow$ constrained_state be a function where EP is an effect predicate. $activ_{EP}$ is a function of* CLPS-B *which returns the constrained state after the activation of the effect k of the operation j.*

$activ_{EP}(\theta_i) = \theta_{i+1}$ can fail if θ_i does not satisfy the pre-condition of EP and if θ_{i+1} does not satisfy the invariant.

Definition 3. *Let compute_children : trace $\rightarrow$ $\{$trace$\}$ be a function that computes all children of a constrained state.*

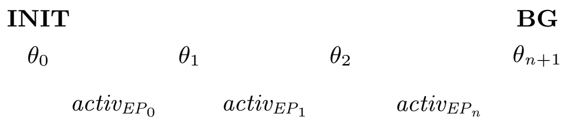


Fig. 6. A trace with forward chaining

$$\begin{gathered} compute\_children(\theta_0, \ldots, \theta_n) \\ = \\ \{T \mid T = \theta_0, \ldots, \theta_n, \theta_{n+1} \wedge activ_{EP}(\theta_n) = \theta_{n+1}\} \end{gathered}$$

The best-first search is run from *compute_preamble*(θ_0) with the function *compute_children* of Definition 3. The cost of T is computed from θ_{n+1}.

Recall that some boundary goals may not be reachable via the available operation (this happens when the invariant is weaker than it should be). By construction every boundary goal satisfies the invariant, which is, of course, a prerequisite for reachability. In addition, the search for the boundary state is bounded during the forward chaining preamble computation, so that the unreachable boundary goals (and unfortunately sometimes some reachable ones) are reported to the test engineer as unreachable. This feedback may suggest ways to strengthen the invariant with a consequence of eliminating such goals.

The traversal of the reachability graph is guided by a cost function, that is used to estimate which states are "closest" to a state verifying the current boundary goal. These numerical measures of closeness are used to decide which states to consider to apply a next operation. If a vertex is a solution, its cost is 0, otherwise its cost is bigger than 0. Obviously, the efficiency of this kind of cost function is relative to the problem and, here, to the formal model. This method is effective if there are no local minima in the reachability graph evaluation regarding the cost function. But, if there are local minima, the algorithm can get lost and fail to find any solution. This point will be illustrated in the working example in the next sub-section.

Simplified Windscreen Wiper Controller Example

The preamble computation is run on the boundary goal BG_1 ($car\_speed = 0$ $\wedge$ $wiper = intermittent$). The trace length is bounded by 3. The cost function is very simple – the number of state variables that cannot be unified with the variables of the boundary goal is counted. The search tree is given in Figure 7. The labels on the edges give the operation name, the effect and the input used to compute the child vertices. Possible solution traces are given by the dotted path. The state cost is in bold in each state.

This animation tree has 40 constrained states. 20 of them have a cost of 1, 18 a cost of 2 and 2 a cost of 0. The algorithm evaluates each constrained state that has a cost of 1 before reaching the state of 0. Because the cost must go up to 2 before going down to 0, the search procedure gets lost in a local minimum (cost 1) before finding the right path.

5 Backward Chaining

The backward chaining procedure is implicitly goal-directed, as one starts with the desired goal. This significantly reduces the search space. Backward chaining makes better use of the effect predicates that were computed during the

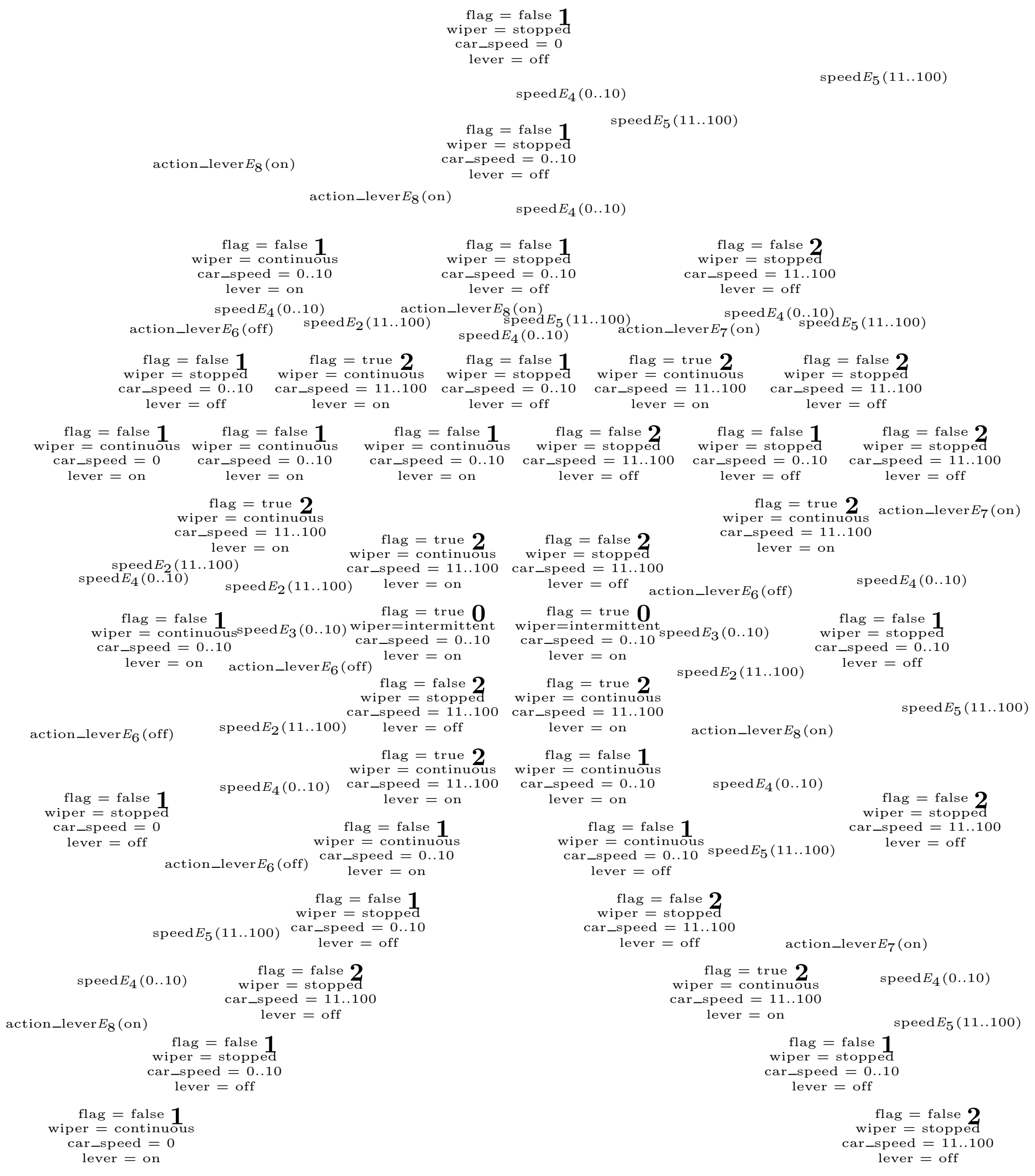


Fig. 7. Search tree in forward chaining

boundary goal generation. Therefore it enables problems with local minima to be avoided.

Before describing the backward chaining algorithm, some preliminary definitions are now provided.

Definition 4. *Let φ_{BG} be the constraint store of the boundary goal BG, and V_{BG} the set of the state variables of BG.*

Definition 5. *Let $\theta_i(\varphi, V)$ be the i^{th} vertex of a trace where φ denotes a constraint store that is consistent, and V denotes a set of state variables to be reached.*

The root of the search tree in the backward chaining procedure is represented by $\theta_0(\varphi_{BG}, V_{BG})$ i.e. by the constraints of the boundary goal that must be reached. For example, the boundary goal $v < 4 \wedge a = 0..3$ gives the vertex $\theta_0(v_0 < 4 \wedge a_0 = 0..3, \{a_0, v_0\})$. v_0 and a_0 are respectively the value representing variables of v and a in θ_0. The variable suffixes allow variables values to be linked to their corresponding vertex.

An effect predicate is made up of constraints being about variables. The variables can be state variables, prime state variables, input variables, output variables or local variables.

For example, in the effect predicate $a = 0..1 \wedge v' = v+1$ with in the invariant $a = 0..10 \wedge v = 0..10$ then the variables a and v are state variables and v' is a prime state variable. At the state θ_i, if the variable v' corresponds to v_i then at the state θ_{i+1}, v corresponds to v_{i+1}. In fact, during an animation, the v_{i+1} is the value of v before the execution of an operation and v_i is the value of v after the execution of the operation.

One needs the function ϕ such as $\phi(\{a_i, b_i, c_i, ..\}) = \{a_{i+1}, b_{i+1}, c_{i+1}, ...\}$ in the next definition.

Definition 6. *Let γ_{EP} : vertex $\rightarrow$ vertex $\cup$ $\{fail\}$ (where EP is an effect predicate) be the function that returns the child of a vertex defined by:*

$$\begin{gathered}\gamma_{EP}(\theta_i(\varphi, V)) \\ = \\ \theta_{i+1}(\varphi \wedge \varphi_{EP} \wedge \varphi_{inv} \wedge (\varphi_{V_{i+1}} = \varphi_{V - V_i^{EP}}), \\ V_{i+1}^{EP} \cup \phi(V - V_i^{EP}))\end{gathered}$$

- *φ_{EP} is the constraint store of the effect predicate EP,*
- *φ_{inv} is the constraint store of the invariant,*
- *φ_V are the constraints on the set of variables V,*
- *V_{i+1}^{EP} is the set of the state variables of the effect predicate EP,*
- *V_i^{EP} is the set of the prime state variables of the effect predicate EP,*
- *V_{i+1} is the set of the state variables of the vertex θ_{i+1}.*

The function γ_{EP} adds all the constraint of the effect predicate EP and of the invariant. The variables which has not been reached ($V - V_i^{EP}$) are unified with their corresponding variables $V(i+1)$ of the vertex θ_{i+1}

The new variables to be reached are $V_{EP(i+1)} \cup (V - V_{EP(i)})$ i.e. all the state variables of EP and the variables that have not been reached.

For example, if EP is $a = 0..1 \wedge v' = v + 1$ with the invariant $a = 0..10 \wedge v = 0..10$ and $\theta_0(v_0 < 4 \wedge a_0 = 0..3, \{a_0, v_0\})$ then :

- φ is $v_0 < 4 \wedge a_0 = 0..3$
- $V = \{a_0, v_0\}$

- φ_{EP} is $a_1 = 0..1 \wedge v_0 = v_1 + 1$
- φ_{inv} is $a_1 = 0..10 \wedge v_1 = 0..10$
- $V - V_i^{EP} = \{a_0, v_0\} - \{v_0\} = \{a_0\}$
- $\varphi_{V_{i+1}} = \varphi_{V - V_i^{EP}}$ is $\varphi_{\{a_1, v_1\}} = \varphi_{\{a_0\}}$ then $a_0 = a_1$
- $V_{i+1}^{EP} = \{a_1, v_1\}$
- $\phi(V - V_i^{EP}) = \phi(\{a_0, v_0\} - \{v_0\}) = \phi(\{a_0\}) = \{a_1\}$
- $V_{i+1}^{EP} \cup \phi(V - V_i^{EP}) = \{a_1, v_1\} \cup \{a_1\} = \{a_1, v_1\}$

Finally, the constraint store of θ_1 is $v_0 < 4 \wedge a_0 = 0..3 \wedge a_1 = 0..1 \wedge v_0 = v_1 + 1 \wedge a_1 = 0..10 \wedge v_1 = 0..10 \wedge a_0 = a_1$ and the new variables to be reached are a_1 and v_1. Simplifying the constraints one obtains:

$$\begin{gathered}\gamma_{EP}(\theta_0(v_0 < 4 \wedge a_0 = 0..3, \{a_0, v_0\})) \\ = \\ \theta_1(v_1 = 0..2 \wedge a_1 = 0..1, \{a_1, v_1\})\end{gathered}$$

To reduce the number of children of a vertex, a pre-condition is added into the function γ_{EP}: only the effects that modified the variables to be reached have to be considered.

Definition 7. *Let Γ_{EP} : vertex $\to$ vertex $\cup$ $\{fail\}$ where EP is an effect predicate, be a function that computes a child of a vertex if EP is applicable:*

$$\begin{gathered}\Gamma_{EP}(\theta_i(\varphi, V)) \\ = \\ \exists\, x . x \in V_i^{EP} \wedge x \in V \mid \\ \theta_{i+1}(\varphi \wedge \varphi_{EP} \wedge \varphi_{inv} \wedge (\varphi_{V_{i+1}} = \varphi_{V - V_i^{EP}}), \\ V_{i+1}^{EP} \cup \phi(V - V_i^{EP}))\end{gathered}$$

Definition 8. *Let compute_children : trace $\to$ $\{trace\}$ be a function that computes all children of a vertex.*

$$\begin{gathered}compute\_children(\theta_0, \ldots, \theta_n) \\ = \\ \{T \mid T = \theta_0, \ldots, \theta_n, \theta_{n+1} \wedge \Gamma_{EP}(\theta_n) = \theta_{n+1}\}\end{gathered}$$

In this case, a trace is a list of vertices. It is straightforward to find the list of associated operations (see figure 8).

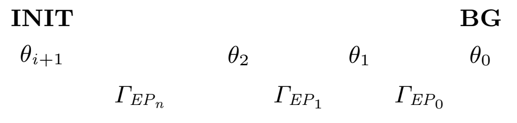


Fig. 8. A trace with backward chaining

The best-first search is run from *compute_preamble*(θ_0) using the function *compute_children* of the Definition 8.

Implementation of the backward chaining

In this part, the Sicstus Prolog implementation of function Γ_{EP} is explained by example. The predicate name which implements this function is `executeEffect-Backward`. Its inputs are a constraint state `CS0`, the variables to be reached `Lvar` and the effect predicate applying `EP`. Its output are the input values `Input` of the effect predicate, the constrained state `CSres` after the application of `EP` and the new variables to be reached `LvarRes`. The Figure 9 shows the implementation of `executeEffectBackward`.

```
executeEffectBackward(CS0,Lvar,EP,Input,CSres,LvarRes):-
   prime(CS0,CS1),
   addConstraintInvariant(CS1,CS2),
   getPrimeStateVarListFromEffect(EP,EPListPrimeVar),
   inclusionIsNotEmpty(Lvar,EPListPrimeVar)*,
   addConstraint(CS2,EP,CS3)*,
   deletePrimeStateVariable(CS3,EPListPrimeVar,CS4),
   unifyPrimeAndStateVariable(CS4,CS5)*,
   getInput(CS5,Input),
   deleteAllExceptStateVariable(CS5,CSres),
   getStateVarListFromEffect(EP,EPListVar),
   computeNewVarToBeReach(EPListVar,Lvar,EPListePrimeVar,LvarRes).
```

Fig. 9. Backward chaining Prolog implementation

A constrained state contains the values of 5 kinds of variables: the state variables, the prime state variables, the input variables, the output variables and the local variables. When a constraint C is added to a variable v in a constrained state CS and v is out of the state, then v is added to the CS. The predicate `addConstraintInvariant` allows to store the invariant on the state variables of a constrained state. The predicate `prime` allows state variables to be moved to primed variables. The predicate `addConstraint` allows to add constraints to a constrained state. Given a constrained state as input the predicate `deletePrimeStateVariable` allows the prime state variables to be deleted. The predicate `deleteAllExceptStateVariable` allows to delete all variables of a constrained state except the state variables. The predicate `unifyPrimeAndStateVariable` allows to unify the prime state variables and their corresponding state variables. The predicate `getInput` allows to obtain the input value of a constrained state.

The predicates `getPrimeStateVarListFromEffect` and `getStateVarList-FromEffect` allow respectively to obtain the list of the prime state variables and the list of the state variables of an effect predicate. Finally, the predicate `computeNewVarToBeReach` allows to compute the new variables to be reached $(V_{i+1}^{EP} \cup \phi(V - V_i^{EP}))$ from the state variables of the effect predicate, the variables to be reached and the prime state variables of the effect predicate.

The predicates with * can fail according to the stored constraints.

The Table 5 shows the evolution of the constrained state during the execution of `executeEffectBackward` from the state $car\_speed = 0 \wedge wiper = intermittent$ and the effect predicate E_4.

CS1 :	car_speed=0 $\wedge$ wiper=intermittent
CS2 :	car_speed'=0 $\wedge$ wiper'=intermittent
CS3 :	car_speed'=0 $\wedge$ wiper'=intermittent $\wedge$ sp=0 $\wedge$ flag=false
CS4 :	wiper'=intermittent $\wedge$ sp=0 $\wedge$ flag=false
CS5 :	wiper=intermittent $\wedge$ sp=0 $\wedge$ flag=false
CSres :	wiper=intermittent $\wedge$ flag=false

Table 5. Values of the constrained states during execution of the effect predicate E_4

Simplified Windscreen Wiper Controller Example

The preamble computation with the backward chaining procedure is run to reach the boundary goal BG_1: *car_speed* $= 0 \wedge$ *wiper* $=$ *intermittent*. So the root of the search tree is: *car_speed* $= 0 \wedge$ *wiper* $=$ *intermittent*. As for the forward chaining procedure, the trace length is bounded by 3 and the cost function is calculated by counting the number of variables that cannot be unified with the initial state. Figure 10 shows the search tree of backward chaining. The labels on the edges give the operation name, the effect and the input used to compute the child vertices. The state cost is given at the top on the right of each state. Possible solution traces are given by the dotted path.

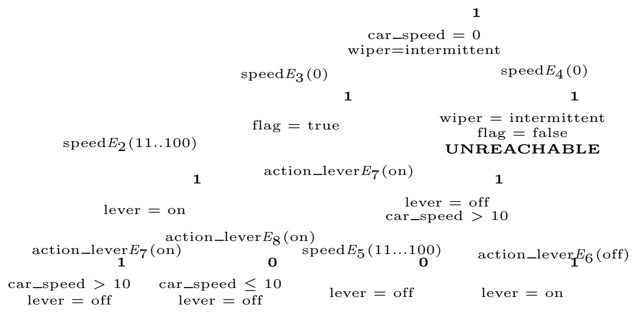


Fig. 10. Search tree in backward chaining

It should be noted that the number of evaluated vertices is dramatically lower than for forward chaining. There are only 9 constrained states in the search tree. One constrained state is detected as unreachable because there is no effect that can be activated from it.

From the root state, only 2 effects (E_3 and E_4) can be applied. The other effects that cannot be applied are:

– E_1 because it does not modify the variables that one wants to reach: `inclusionIsNotEmpty(CSListStateVar, EPListStateVar)` fails.
– E_2, E_6, E_7, E_8 because,they do not put the variable *wiper* at the right value (*continuous* or *stopped* instead of *intermittent*): `addConstraint(CS1, EP, CS2)` fails,
– E_5 because it puts *car_speed* to *sp*, *sp* should be 0 but it is greater than 10: `addConstraint(CS1, EP, CS2)` fails.

The other vertices are computed the same way. The main reasons for the better convergence of the backward chaining procedure are:

– Effects that do not modify the variables of the vertex are not considered in the current search.
– The application of an effect can fail on the before and the after part of the effect whereas with the forward chaining procedure, the application of an effect can only fail on the before part of the effect. Thus, a state has a greater chance to produce fewer children.
– The before part of the effects is exploited and guides the search by computing all effects that allow to reach a constraint state and so on until the initial state.

6 Experimental Results

The BZ-TT technology has been applied to several industrial applications for smart cards and transport systems. In particular, tests have been automatically generated from a real windscreen wiper controller application [25] in partnership with the PSA Peugeot Citroën company. The test generation process started by formalizing the technical requirements with the B notation. The B abstract machine ran to 25 pages and contained 25 operations and 15 state variables. This formal model represents 380 160 000 states. This application showed the limitations of the forward chaining procedure in the presence of many local optima. The cost function was the one used in the previous example. In the corresponding search tree, there were a lot of local minima. 526 boundary goals were identified, of which 254 reachable and 272 were not (confirmed by informal checking). The experimental results are presented in Table 6 (the preamble computation was run using a 1500 vertex limit and a trace depth of 10).

	Number of BG	Average number of vertices evaluated
Backward chaining limit : 1500, depth : 10	235 found	290
	219 unreachable_at_this_vertex_number	1511
	70 unreachable_at_this_depth	61
	12 unreachable	43
Forward chaining limit : 1500, depth : 10	155 found	324
	371 unreachable_at_this_vertex_number	1505
	0 unreachable_at_this_depth	0
	0 unreachable	0

Table 6. Windscreen wiper speed controller experimental results

Backward chaining reached 92.5% of the reachable boundary goals whereas forward chaining reached only 61%. Moreover backward chaining found certain boundary goals to be unreachable where forward chaining did not.

All of the boundary goals reached by forward chaining were also reached by backward chaining. The 22 missing boundary goals are reached by backward chaining with a vertex limit around 130 000. Table 7 presents the statistical results about the number of evaluated vertices for each boundary goal with the backward chaining procedure.

The average number of vertices evaluated is 3 766.64, but in fact, the measures of spread show that 75% of boundary goals were found with less than 411 evaluated vertices. Otherwise, the search gets lost on a plateau (the vertices which have the lower cost are very numerous). This problem should be solved by improving the cost function.

Smallest value	0
Largest value	130 640
Mean	3766.64
Median (Q_2)	308
Range	130 640
Lower quartile (Q_1)	175
Upper quartile (Q_3)	411
Interquartile range	236
Variance (S^2)	290 416 857
Standard deviation (S)	17 041

Table 7. Statistics relating to the number of vertices evaluated when using backward chaining

In the industrial case study backward chaining was more effective than forward chaining. More generally, this is the case when the formal model has many operations with weak pre-conditions that allow their execution from a large number of reachable states (in particular from the initial state of the machine). When also these operations modify a restricted number of state variables, this gives a smaller number of choice points for backward chaining in comparison of forward chaining.

7 Conclusion

This paper has described the advantages of a backward chaining procedure over a forward one for the computation of test case preambule sequences. The underlying Finite State Automaton is not built to compute test sequences: the preamble computation makes it possible to find a trace from the initial state to a state that satisfies particular constraints (the boundary goal). Moreover, during the reachability graph traversal, the CLPS-B solver computes a constraint store representing a set of valuated concrete states. It corresponds to a symbolic animation of the formal model and helps to master the combinatorial explosion of the search.

This study is part of the research field of using CLP techniques for software verification. Over the last few years, constraint technology has been used for various purposes, such as model checking e.g. [26], formal model animation e.g. [27] and automated test generation that is either code-based e.g. [28] or specification-based e.g. [29, 7]. In particular, in specification-based test generation, the use of constraint technologies makes it possible to symbolically execute the formal model without constructing the underlying Finite State Automaton.

Improving the preamble computation provides better scalability of such model-based test generation tools such as BZ-TT. In spite of a very simple cost function, backward chaining finds 31% more reachable boundary goals than forward chaining.

The BZ-TT environment has been used for half a dozen industrial applications between 1999 and 2003 in the area of smart card software, automotive embedded systems and bank electronic payment. All these real size formal models have large combinatorial state spaces. For these applications, between 70% and 90% effect coverage was achieved using our preamble computation algorithm. The boundary coverage criteria also helps the validation engineer to efficiently drive the automated generation process.

The BZ-TT technology is now mature enough to be used in industrial setting for animation and model-based testing. So, the technology is currently being transferred to a start-up company, LEIRIOS technologies [30], who will enhance it to allow its use by commercial users, market it and use it for outsourced testing projects. The commercial tool, called LEIRIOS Test Generator, will also be able to handle Statecharts Statemate [31] and UML (class diagrams with OCL pre/post specifications of methods) in the same uniform manner.

Acknowledgment

The authors wish to acknowledge Mark Utting from The University of Waikato (New-Zealand) for his detailed reading and helpful suggestions in the preparation of an earlier version of this paper and John Clark from The University of York (United Kingdom) for his detailed reading, which substantially improved the presentation.

This work has been partially funded by the company PSA Peugeot Citroën.

References

[1] J.M. Spivey. *The Z notation: A Reference Manual.* Prentice-Hall, 2^{nd} edition, 1992. ISBN 0 13 978529 9.

[2] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 2^{nd} edition, 1990.

[3] J-R. Abrial. *The B-BOOK: Assigning Programs to Meanings.* Cambridge University Press, 1996. ISBN 0 521 49619 5.

[4] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the International Conference on Formal Methods Europe (FME'93)*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, April 1993.

[5] H.M. Hörcher and J. Peleska. Using formal specifications to support software testing. *Software Quality Journal*, 4(4):309–327, 1995.

[6] R. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification and Reliability*, 7:19–33, 1997.

[7] L. Van Aertryck, M. Benveniste, and D. Le Metayer. CASTING: a formally based software test generation method. *In 1^{st} IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 99–112, 1997.

[8] S. Behnia and H. Waeselynck. Test criteria definition for B models. In *Proceedings of the World Congress on Formal Methods (FM'99)*, volume 1708 of *LNCS*, pages 509–529, Toulouse, France, 1999. Springer Verlag.

[9] R. Hierons, S. Sadeghipour, and H. Singh. Testing a System specified using Statecharts and Z. *Information and Software Technology*, 43(2):137–149, 2001.

[10] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *Proceedings of the International Conference on Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 21–40, Copenhagen, Denmark, July 2002. Springer Verlag.

[11] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B – A constraint solver for B. In *Proceedings of the ETAPS'02 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 188–204, Grenoble, France, April 2002. Springer Verlag.

[12] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02)*, pages 105–120, Brnö, Czech Republic, August 2002. INRIA Technical Report.

[13] B. Legeard and F. Peureux. Generation of functional test sequences from B formal specifications - Presentation and industrial case-study. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, pages 377–381, San Diego, USA, November 2001. IEEE Computer Society Press.

[14] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11.11 standard case-study. *The Journal of Software Practice and Experience*, 34(10):915 – 948, 2004.

[15] F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation: the Java Card transaction mechanism case study. In *Proceedings of the International Conference on Formal Methods Europe (FME'03)*, volume 2805 of *LNCS*, pages 778–795, Pisa, Italy, September 2003. Springer Verlag.

[16] F. Bouquet, S. Colin, and B. Legeard. Génération automatique de scénarii de test à partir de modèles B - STE visibilité générique (confidential). Technical Report TR-02/02, LIFC - University of Franche-Comté, 2002.

[17] A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In *Proceedings of the CONCUR'01 Workshop on Formal Approaches to Testing of Software (FATES'01)*, pages 47–60, Aalborg, Denmark, August 2001. BRICS.

[18] F. Bouquet, B. Legeard, and F. Peureux. A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer*, 6(2):143–157, August 2004.

[19] C. Gervet. Interval Propagation to reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(2):191–246, 1997.

[20] Swedish Institute of Computer Sciences. *SICStus Prolog 3.8.7 manual documents*, October 2001. http://www.sics.se/sicstus.html.

[21] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th Symposium on Principles of programming Languages (POPL'87)*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[22] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[23] B. Legeard, F. Peureux, and M. Utting. Controlling Test Case Explosion in Test Generation from B Formal Models. *The Journal of Software Testing, Verification and Reliability*, 14(2):81–103, 2004.

[24] ISO. Information Processing Systems, Open Systems Interconnection. *OSI Conformance Testing Methodology and Framework – ISO 9646*, 1998.

[25] S. Colin. *Génération automatique de scénarii de test - Application de la méthode BZ-TT à un cas industriel - Amélioration du calcul du préambule.* Master's thesis, LIFC - University of Franche-Comté, 2002.

[26] G. Delzanno and A. Podelski. Model Checking in CLP. In *Proceedings of the ETAPS'99 International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 223–239, Amsterdam, The Netherlands, March 1999. Springer Verlag.

[27] W. Grieskamp. A computation model for Z based on concurrent constraint resolution. In *Proceedings of the International Conference of Z and B Users (ZB'00)*, volume 1878 of *LNCS*, pages 414–432, September 2000.

[28] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In *Proceedings of the First International Conference on Computational Logic (CL'00)*, pages 399–413, London, UK, July 2000. Springer Verlag.

[29] B. Marre and A. Arnould. Test Sequence generation from Lustre descriptions: GATEL. In *Proceedings of the 15th International Conference on Automated Software Engineering (ASE'00)*, pages 229–237, Grenoble, France, 2000. IEEE Computer Society Press.

[30] The Leirios web site. http://www.leirios.com, 2005.

[31] F. Bouquet, B. Legeard, and F. Lebeau. Test-Case and Test-Driver Generation for Automotive Embedded Software. In *Proceedings of the 5th International Conference on Software Testing (ICS-Test'04)*, pages 37 – 53, Dusseldorf, Germany, April 2004. Software and Systems Quality Conferences.