Checking temporal properties on state based formal specification: application to railway level crossing

David MENTRÉ
Mitsubishi Electric MERCE France
1 allée de Beaulieu, CS 10806
35708 Rennes Cedex 7, France
Email: d.mentre@fr.merce.mee.com

***Abstract*—B Method, a formal method widely used in railway industry to specify and implement safety critical software, lacks expression and verification of temporal properties. Those properties are needed to check system evolution over several successive states. We propose in this paper an approach to check temporal properties of B models using Model Checking techniques. More specifically, we transform B models into SPIN models and check temporal properties expressed in Linear Temporal Logic (LTL) on them. We apply our approach on the B model of a railway level crossing software system, checking that after closing barriers should eventually open.**

I. INTRODUCTION

Software is nowadays widely used in safety critical parts of transportation systems, from car engines to Automatic Train Protection systems. In order to improve the quality and maintainability of such software, Formal Methods are used [1]. Using Mathematics, they allow to design and produce software with the proof that some parts fulfil a given requirement in all possible cases. Amongst those formal methods, the B Method [2] is widely used in the railway industry [3], [4]. A B Method formal model describes a system through its states and state transformation operations (using sets, relations, integers and boolean logic). As shown in section III, invariants are used to express the properties that the software system should fulfil at any time, i.e. before and after the update of system state by an operation. An abstract B model can be refined into a concrete, implementable, computer program called a B0 model. At each refinement step, the B Method requires that some Proof Obligations must be proved to ensure that invariants specified at the abstract level are still ensured by the refinements.

Invariants allow to specify safety properties that must be ensured *at any time*. However, using invariants it is not possible to express properties valid on the *successive* states of the system. For example, if the B Method is used to specify and implement the software controlling the barriers of a railway level crossing, an invariant can easily express that, at any time, if a train is at the crossing, barriers should be closed. However one also wants to ensure that after the barriers are closing they will eventually open again in any possible future. Please notice that we propose to check properties on successive system states (e.g. state B is always reachable from state A) and not real time properties (it takes 10 seconds to go from A to B).

In this paper, we propose a simple approach to check such temporal properties in B formal models using the SPIN model checker [5] described in section IV. The main contribution of this paper, more precisely seen in section V, is to automatically transform a B0 level model (i.e. an implementable B program) into a Promela model suitable to be extended and checked using the SPIN model checking tool. After automatic conversion, the model is manually extended with a short description of the environment and the formulation of properties to check as described in section VI. Once done, the SPIN model checker is applied to check that the property is valid in all possible successive states of the model or to provide a counter example if this not the case. In order to illustrate our approach, we apply our proposal on an example of a software controlling a railway level crossing described in section II. We compared this work with related work in section VII and finally conclude in section VIII.

II. THE RAILWAY LEVEL CROSSING EXAMPLE

In order to illustrate our approach, we use the example of a software controlling the barriers of a railway level crossing [6], [7]. This simplified railway level crossing is made of one unidirectional railway with two detectors ADC and BDC before and after the crossing. The section of rails between the two detectors is called the *warning section*. Any train within this section should activate the bells and lights of the crossing in order to warn users.

The software controlling the crossing is an automaton build upon four states (*open*, *closing*, *closed* and *opening*) and nine transitions between those states, as shown in figure 1. Initially, the barriers are opened. When a first train enters the warning section, the barriers start to close (*closing* state) until they are closed (*closed* state). They should close in at most 15 seconds. The barriers start to open (*opening* state) once 20 seconds have elapsed in the *closed* state and there is no train in the warning section. If during barrier opening a new train enters the warning section, a new closing cycle is initiated.

A safety property of the system is that in all three states *closing*, *closed* and *opening*, the bells and lights should be activated. Another property is that if there is a train in the warning section, then the barriers should not be in the *open* state. A third property is that if the automaton enters the *closing* state and the warning section is empty (i.e. there is

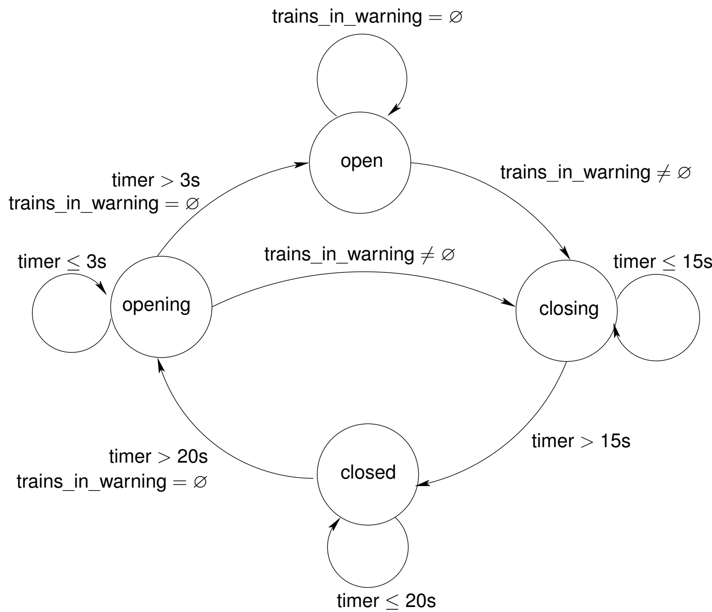

Fig. 1. Railway level crossing automaton

no train in it), then the automaton should eventually reach the *open* state. While the two first properties can be expressed as system invariant, the third properties requires a temporal logic to be expressed.

III. Formal software development with the B Method

We have developed a model of this railway level crossing example using the B Method. This model contains both an abstract description of the barrier automaton and its implementation at B0 level. This implementation can be translated into compilable C code and the resulting program can be executed.

A B Method model is organised around Abstract Machines. Each machine is either a part of the model (e.g. lights or barrier automaton) or a machine made in order to be used by another machine so as to simplify the proof of the model. Each machine is made of variables and invariants. The variables contain the state of the machine and the invariants describe static relationships between the variables.

For example, in the following Signals machine, three boolean variables are used to represent the lights, the bells and the abstract signals. An invariant expresses that signals are activated if and only if lights and bells are activated.

```
MACHINE Signals

VARIABLES signals_on, lights_on, bells_on

INVARIANT
  signals_on ∈ 𝔹 ∧ lights_on ∈ 𝔹 ∧ bells_on ∈ 𝔹 ∧

  signals_on = TRUE ⇔ (lights_on = TRUE ∧ bells_on = TRUE)
```

In an Abstract Machine, state transformation are represented using abstract operations that transform a state to another one. For example in following code, after an initialisation step, two operations are defined: activate_signals and deactivate_signals to respectively activate and deactivate the signals in the model. After initialisation step and before and after each operation abstract machine's invariants should be ensured.

```
INITIALISATION
  signals_on := FALSE || lights_on := FALSE || bells_on := FALSE

OPERATIONS
  activate_signals =
  BEGIN
    signals_on := TRUE || lights_on := TRUE || bells_on := TRUE
  END;

  deactivate_signals =
  PRE signals_on = TRUE
  THEN
    signals_on := FALSE || lights_on := FALSE || bells_on := FALSE
  END
```

The B Method is based on *refinements*. The defined abstract machines are refined into more detailed machines until having enough details to be implemented. Those so called B0 machines are using simple, implementable, types like 32-bits integers or arrays. The correctness of a refinement with respect to its refined machine is ensured through Proof Obligations that are automatically generated by a tool following the B Method formalism. Those Proof Obligations should be manually or automatically proved.

For example, the previous Signals machine can be refined into an implementable Signals_i machine. The two operations are implemented through function calls to operations defined in other imported abstract machines Lights and Bells.

```
IMPLEMENTATION Signals_i REFINES Signals

IMPORTS Lights, Bells

OPERATIONS
  activate_signals =
  BEGIN
    start_lights ; start_bells
  END;

  deactivate_signals =
  BEGIN
    stop_lights ; stop_bells
  END
END
```

As an example of Proof Obligation to check, one needs to prove the following one for Signals_i implementation: signals_on = **TRUE** ⇒ lights_on = **TRUE**. It ensures that under the precondition of the deactivate_signals operation (signals_on = **TRUE**), the pre-condition of the called stop_lights operation (lights_on = **TRUE**) is always fulfilled.

To formalise and implement our case study, we have defined nine abstract machines and their implementations (suffixed with _i) according to the architecture given in figure 2. The Main_loop machine contains the body of the program and is called at a regular pace (each 100 ms). At each loop iteration, it reads the detectors, it computes the arrival or departure of trains as well as the new automaton state and finally triggers the actuators (e.g. switch on lights). The Barrier_context machine is a so-called "context machine" used to define constants used by other abstract machines. The Warning_section machine is the abstraction of the warning section and its detectors. The Barrier and Barrier_transitions machines contain the core of the program, the computation of the barrier automaton transitions.

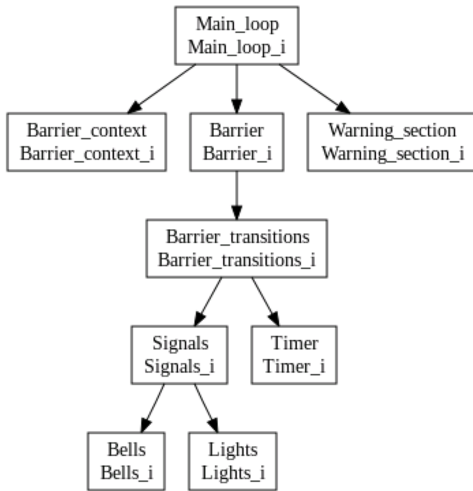

Fig. 2. Architecture of railway level crossing in B Method

```
IMPLEMENTATION Barrier_i REFINES Barrier
 ...
OPERATIONS
    advance_barrier_automaton =
    VAR st, warning_empty, rt IN
        st ← read_status;
        warning_empty ← is_warning_section_empty;
        rt ← read_remaining_time;
        CASE st OF
            EITHER open THEN
                IF warning_empty = FALSE THEN open_to_closing END
            OR closing THEN
                IF rt = 0 THEN closing_to_closed END
            OR closed THEN
                IF rt = 0 ∧ warning_empty = TRUE THEN closed_to_opening END
            OR opening THEN
                IF rt = 0 ∧ warning_empty = TRUE THEN opening_to_open
                ELSIF warning_empty = FALSE THEN opening_to_closing END
            END
        END
    END
END
```

Fig. 3. advance_barrier_automaton operation

Finally, the Timer machine is used to measure time and the Signals, Bells and Lights machines implement the signals used to warn users.

Invariants are used to state security properties that our program should ensure. For example machine Main_loop_i contains the following invariant that states that if the warning section (represented here by an abstract set trains_in_warning) is not empty, then signals should be activated.

INVARIANT trains_in_warning $\neq \varnothing \Rightarrow$ signals_on = **TRUE**

Due to space constraint, we cannot give the full detailed of the abstract machines. We will therefore only describe the machines containing the automaton defined in figure 1. Moreover we will focus on their implementation and not their abstract specification as former machines are going to be re-used later in this article.

The first Barrier machine contains in its implementation an operation advance_barrier_automaton as defined in figure 3. This machine reads the current barriers status (*open*, *closing*, ... in variable st), if the warning section is empty or not (in warning_empty) and the current remaining time in the timer (in rt). From those information, it computes the next state with a **CASE** construct. If a state transition can occur, a corresponding operation, e.g. open_to_closing operation to go from the *open* state to the *closed* state, is called to do the state transformation for this transition.

```
IMPLEMENTATION Barrier_transitions_i REFINES Barrier_transitions
 ...
OPERATIONS
    open_to_closing =
    BEGIN
        status := closing; set_timer(15000); activate_signals
    END;

    closing_to_closed =
    BEGIN
        status := closed; set_timer(20000)
    END;
 ...
```

Fig. 4. Two example of automaton state transformation as defined in Barrier_transitions

Such state transformation operations are defined in the abstract machine Barrier_transitions as shown in figure 4. Those operations are simply changing the current status through an assignment, initialising a new timer and activating or de-activating signals through function calls to activate_signals or deactivate_signals.

The Proof Obligations made mandatory by the B Method ensures that security invariants are fulfilled by our program. But using invariants, it is only possible to describe properties valid at any state of the program. We should use a different approach to check properties valid over successive states of the program, i.e. temporal properties.

IV. Model checking with SPIN

In order to check temporal properties, one can use Model Checking approaches [8]. They are various but they all use a model of concurrent entities defined using a modelling language. Those entities are exchanging information through shared variables, channels or signals. The state space of this model is then explored in order to check the temporal properties to verify over successive states of the system.

In this study, we are using the SPIN [5] model checker. To check temporal properties, SPIN does an exhaustive state space exploration: starting from the initial system state, it enumerates all the reachable states, back-tracking in case a state has already been seen. On each path starting at the initial state, SPIN checks that validity of stated temporal properties.

The SPIN's modelling language Promela is a Pascal-like language that includes non-determinism. Promela has "`if`" testing and "`do`" looping constructs. In both cases, if several `if` or `do` statements can be executed at construct entrance, one is chosen non-deterministically. By default, all testing statements are blocking, i.e. a "`(variable == value)`" statement will stop model's execution until the variable reaches the specified value. Both `if` and `do` constructs support an "`else`" statement

Definition

$\mathcal{T}$[**VALUES** A = v]	$\triangleq$	**#define** A (v)
$\mathcal{T}$[**SETS** S = {a, b}]	$\triangleq$	**mtype** = { a, b }
$\mathcal{T}$[**VARIABLES** a, b **INVARIANT** a $\in \mathbb{B} \wedge$ b $\in$ INT **INITIALISATION** a := **FALSE**; b := 0]	$\triangleq$	**bool** a = **false;** **int** b = 0;

Expression

$\mathcal{T}$[identifier]	$\triangleq$	*machine*__identifier
$\mathcal{T}[n], n \in \mathbb{Z}$	$\triangleq$	n
$\mathcal{T}$[**TRUE**]	$\triangleq$	**true**
$\mathcal{T}$[**FALSE**]	$\triangleq$	**false**
$\mathcal{T}[a = b]$	$\triangleq$	$\mathcal{T}[a]$ == $\mathcal{T}[b]$
$\mathcal{T}[a \neq b]$	$\triangleq$	$\mathcal{T}[a]$!= $\mathcal{T}[b]$
$\mathcal{T}[\neg a]$	$\triangleq$	! $\mathcal{T}[a]$
$\mathcal{T}[a \leq b]$	$\triangleq$	$\mathcal{T}[a]$ <= $\mathcal{T}[b]$
	$\vdots$	
$\mathcal{T}[a \wedge b]$	$\triangleq$	$\mathcal{T}[a]$ && $\mathcal{T}[b]$
$\mathcal{T}[a \vee b]$	$\triangleq$	$\mathcal{T}[a]$ || $\mathcal{T}[b]$

Operation definition

$\mathcal{T}$[**OPERATIONS** r $\leftarrow$ op(a) **BEGIN** body **END**]	$\triangleq$	**inline** *machine*__op(r, a) { *type* tmp = $\mathcal{T}$[body]; r = tmp; }
$\mathcal{T}$[**PROMOTES** op]	$\triangleq$	**#define** op op_{source}

Operation body

$\mathcal{T}$[**skip**]	$\triangleq$	**skip**
$\mathcal{T}[S;S']$	$\triangleq$	$\mathcal{T}[S]$ **;** $\mathcal{T}[S']$
$\mathcal{T}$[V := e]	$\triangleq$	V = $\mathcal{T}[e]$**;**
$\mathcal{T}$[**IF** c **THEN** S **ELSE** S' **END**]	$\triangleq$	**if** :: c -> $\mathcal{T}[S]$ :: !c -> $\mathcal{T}[S']$ :: **else** -> **skip** **fi**
$\mathcal{T}$[**CASE** v **OF** **EITHER** v_1 **THEN** S_1 **OR** v_2 **THEN** S_2 ... **END**]	$\triangleq$	**if** :: v == v_1 -> $\mathcal{T}[S_1]$ :: v == v_2 -> $\mathcal{T}[S_2]$ ... :: **else** -> **skip** **fi**
$\mathcal{T}$[r $\leftarrow$ op(a)]	$\triangleq$	r = *machine*__op(a)
$\mathcal{T}$[**VAR** a **IN** body]	$\triangleq$	*type* a; $\mathcal{T}$[body]

Fig. 5. B0 to Promela transformation rules

that is executable in case all of other construct statements are blocking, thus allowing to model the testing and looping constructs used in regular programming languages.

V. B0 TO PROMELA TRANSFORMATION

In this paper, we transform B Method models at B0 level, i.e. implementable programs, into Promela models. We have chosen the B0 level because it seems easier to convert to Promela than more abstract levels (e.g. absence of indeterministic assignation, values given for all constants, no multiple assignment, ...). We have not evaluated yet if a similar work could be done for more abstract B models.

```
inline Barrier__advance_barrier_automaton() {
  mtype st;
  Barrier_transitions__read_status(st);
  bool warning_empty;
  Warning_section__is_warning_section_empty(warning_empty);
  int rt;
  Timer__read_remaining_time(rt);

  if
    :: st == open ->
       if
         :: warning_empty == false
            -> Barrier_transitions__open_to_closing();
         :: else -> skip
       fi
    :: st == closing ->
       if
         :: rt == 0
            -> Barrier_transitions__closing_to_closed();
         :: else -> skip
       fi
    :: st == closed ->
       if
         :: rt == 0 && warning_empty == true
            -> Barrier_transitions__closed_to_opening();
         :: else -> skip
       fi
    :: st == opening ->
       if
         :: rt == 0 && warning_empty == true
            -> Barrier_transitions__opening_to_open();
         :: else ->
            if
              :: warning_empty == false
                 -> Barrier_transitions__opening_to_closing();
              :: else -> skip
            fi
       fi
    :: else -> skip
  fi
}
```

Fig. 6. Promela's version of advance_barrier_automaton operation

The B0 to Promela transformation is done in a systematic way following $\mathcal{T}$ function given in figure 5. Regarding definition, valued constants are translated into **#define** constructs. Enumerated sets are transformed into Promela's **mtype** but deferred sets are not converted as their values are not used. For variables, we are using their typing in **INVARIANT** clause and their initialisation to chose the corresponding Promela's type (**bool** for $\mathbb{B}$, **int** for NAT and INT) and initial value.

Regarding expression, the translation is straightforward and purely syntactical. Arithmetic expressions are kept as is. Identifiers are prefixed with their machine name and defined as global variable in Promela model.

For operation definition, Promela does not support real procedure call but only macro-like rewriting. Moreover, returned parameters are not supported by Promela, therefore we introduce a return parameter `r` as first parameter into the corresponding **inline** definition in Promela. Within operation body a temporary variable `tmp` of adequate type is used (determined using B0's type inference). When an operation is promoted from an imported B0 machine, we simply **#define**

```
active proctype train() {
  /* trigger first detector */
  Warning_section__adc_state = true;
  (Warning_section__adc_state == false); /* wait */

  /* trigger second detector */
  Warning_section__bdc_state = true;
  (Warning_section__bdc_state == false); /* wait */
}

init {
end: do
       :: d_step {

          /* call Promela's version of B development */
          Main_loop__do_loop();

          /* update detector status */
          if
            :: Warning_section__adc_state == true ->
               Warning_section__adc_state = false;
            :: else -> skip
          fi;
          if
            :: Warning_section__bdc_state == true ->
               Warning_section__bdc_state = false;
            :: else -> skip
          fi;
       }
       :: else -> break
  od;
}
```

Fig. 7. Manually written part in the Promela model

its name as an alias to the name with which the function was previously using, as all variables and **inline** definitions are prefixed with their machine name (in order to avoid an eventual name clash).

In operation body, we use the adequate Promela construct corresponding to B0's one. For **IF** and **CASE** constructs, we should take care to introduce an **else** option so the evaluation of the whole construct is not blocked if all options of **if** or **do** constructs are not executable. For **VAR** variable definition, we need to use B0 type inference to chose the proper *type* definition that should be defined in Promela's version.

Currently, we do not handle the **WHILE** loop construct, **ELSIF** case and the use of arrays, but their translation should be straightforward considering the translations already made. Once all translations are automatically applied, we obtain a Promela model like the one given in figure 6 for Promela's version of advance_barrier_automaton operation originally defined in figure 3.

VI. Manual steps and verification

The application of previous translation produces a Promela model of the software developed using the B Method. However other parts of the software might be missing as well as the description of software's environment. Therefore we need to extend the automatically generated Promela model with a manually written part.

In our particular case, the B development is integrated into an infinite loop which calls the loop body at regular interval each 100 ms. Thus, we add to the Promela's model a **do** construct that calls the `Main_loop__do_loop()` **inline** function into the generated Promela model. The **inline** function call is included in a **d_step** Promela construct that ensures the whole loop body is atomic. This reflects the real use of the software (each loop iteration is an atomic-like operation where detectors are read, output computed and actuators changed) and decreases the number of states in the generated model. We also need to add the model of the detectors. In our particular case, we add some Promela's code to set back the detectors to the **false** state if they have been activated. All of this manually added code constitutes the body of the **init** process of figure 7.

We also need to model external entities like trains. For a simple verification, we just model the crossing of a train with the addition of an independent process **active proctype** `train()`. This process synchronises itself with the **init** process by reading globally accessible variables of the latter, e.g. in construct `Warning_section__adc_state ==` **false** it waits for the first detector to return to the **false** state.

Last but no least, we need to express temporal properties to check. For example we can check that if the barriers are closing, they should eventually open, which is expressed more formally by the temporal formula:

$$\Box(\text{Barrier_transitions__status} = \text{closing} \Rightarrow \Diamond(\text{Barrier_transitions__status} = \text{open}))$$

This formula means that in all possible states ($\Box$), if the barriers are *closing* then they eventually *open* in the future ($\Diamond$). One should notice that we can express properties over successive states but not real-time properties, e.g. barrier should close in less than 15 seconds.

Using SPIN's `-f` option on formula `!([](closing_barrier-> <>open_barrier))`, it is possible to convert this formula into a SPIN's **never** automaton, i.e. a state observer modelling property that should *never* occur. This automaton is included into the Promela model with definitions **#define** `closing_barrier (Barrier_transitions__status == closing)` and **#define** `open_barrier (Barrier_transitions__status == open)`.

After using SPIN to generate the verifier and call it using the `-f` fairness option[1], we can check that the above property is satisfied. 32365 transitions are made for a maximum depth of 1561 states (2.7 megabytes of memory used, verified in 0.03 seconds on a regular Athlon AMD64 at 2 Ghz).

If we add an error in the B model, for example by removing the call to opening_to_open operation in Barrier_i machine, SPIN finds the error. The generated trail shows that SPIN is stuck in the *opening* state without ability to reach the *open* state. One should notice that such kind of error cannot be found only using the B Method.

VII. Related works

Railway level crossing in its generic form [9] is a classical study in the formal methods research community. In this paper,

[1] The fairness option is needed as in this particular case we don't want to check situations where some processes would be infinitely blocked, e.g. a train passing the first detector but never the second one.

we take as example a simplified version [6], [7] with a single unidirectional railway.

The idea of model checking state based formal specification is not new, for example Smith and Wildman model check Z specifications [10]. Regarding the B Method, several proposals have been made to extend the B Method for timed properties, either for classical B Method used in this paper [11], [12], [13] or for Event B formalism [14], [15]. In comparison, the main contribution of this paper is a simpler and practical proposal supported by widely available tools to check temporal properties. We do not modify the current semantics of B and current tools like Atelier B can be used as is.

The work around ProB [16] is probably the most similar to ours with the extension of this tool to check LTL formula [17]. The main difference is that ProB is able to work on abstract B models while our approach is currently only valid for B0 level models. In [18], it is claimed that ProB is as efficient as SPIN to check temporal properties, while writing B models being much more easier than Promela models. We currently do not have any evidence to assert nor invalidate this claim. However the automatic generation of most of the Promela models we propose makes Promela model writing obviously much easier for a comparison.

VIII. Conclusion

In this paper, we show a simple but effective technique to transform a B Method model into a Promela model on which one can check temporal properties. This is useful to extend verification capabilities of the B Method to temporal properties *without the need to build a dedicated model* (and thus avoiding model maintenance issues). This approach is applied on the B model of a piece of software controlling a railway level crossing. We have made an implementation of the presented approach using the B Compiler bcomp [19].

This paper is a first step and several points could be refined. Firstly, we need to complete the translation scheme for the **WHILE** construct, **ELSIF** case and the use of arrays. Secondly we need to give semantics to this transformation, linking B Method and Promela semantics.

A further extension of this work could be to check real-time properties (e.g. the barrier should close in less than 15 seconds) by generating timed automata for the Uppaal [20] model checker. This would necessitate more work as a timed automaton should be associated to each B language construct. Another possible extension of this work would be to apply such transformation to more abstract B models in order to check them as done in ProB tool. A third extension of this work would be to still start from B0 models but to parameterise the transformation to Promela in order to reduce the state space of the generated model, for example by bounding the values of some variables.

Acknowledgment

The author would like to thank Georges Mariano and the anonymous reviewers for providing helpful comments on a previous draft of this paper.

References

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, “Formal methods: Practice and experience,” *ACM Computing Surveys*, vol. 41, no. 4, pp. 1–36, 2009.

[2] J.-R. Abrial, *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[3] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier, “Météor: A successful application of B in a large project,” in *World Congress on Formal Methods*, ser. Lecture Notes in Computer Science, J. M. Wing, J. Woodcock, and J. Davies, Eds., vol. 1708. Springer, 1999, pp. 369–387.

[4] F. Badeau and A. Amelot, “Using B as a high level programming language in an industrial project: Roissy VAL,” in *ZB*, ser. Lecture Notes in Computer Science, H. Treharne, S. King, M. C. Henson, and S. A. Schneider, Eds., vol. 3455. Springer, 2005, pp. 334–354.

[5] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003.

[6] S. Liu, M. Asuka, K. Komaya, and Y. Nakamura, “An Approach to Specifying and Verifying Safety-Critical Systems with Practical Formal Method SOFL,” in *Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 98)*, 1998, pp. 100–114.

[7] ——, “Applying SOFL to Specify A Railway Crossing Controller for Industry,” in *2nd IEEE Workshop on Industrial- Strength Formal Specification Techniques (WIFT 98)*, 1998, pp. 16–27.

[8] R. Jhala and R. Majumdar, “Software model checking,” *ACM Computing Surveys*, vol. 41, no. 4, pp. 1–54, 2009.

[9] C. L. Heitmeyer, B. G. Labaw, and R. D. Jeffords, “A benchmark for comparing different approaches for specifying and verifying real-time systems,” in *Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software*. IEEE Computer Society Press, 1993.

[10] G. Smith and L. Wildman, “Model checking Z specifications using SAL,” in *International Conference of Z and B Users (ZB 2005), volume 3455 of LNCS*. Springer-Verlag, 2005, pp. 87–105.

[11] S. Schneider and H. Treharne, “Communicating B Machines,” in *ZB*, ser. Lecture Notes in Computer Science, D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, Eds., vol. 2272. Springer, 2002, pp. 416–435.

[12] S. Colin, G. Mariano, and V. Poirriez, “Duration Calculus: A Real-Time Semantic for B,” in *ICTAC*, ser. Lecture Notes in Computer Science, Z. Liu and K. Araki, Eds., vol. 3407. Springer, 2004, pp. 431–446.

[13] M. Rached, J.-P. Bodeveix, M. Filali, and O. Nasr, “A Timed B Method for Modelling Real Time Reactive Systems,” in *SEEFM05. 2nd South-East European Workshop on Formal Methods, Ohrid (Macédoine), 18/11/05-20/11/05*. http://www.seerc.info: South-East European Research Center (SEERC), novembre 2005, pp. 181–195.

[14] S. Chouali, J. Julliand, P.-A. Masson, and F. Bellegarde, “PLTL Partitionned Model-Checking for Reactive Systems under Fairness Assumptions,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 4, no. 2, pp. 267–301, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1067915.1067918

[15] H. R. Barradas and D. Bert, “Propriétés dynamiques avec hypothèses d'équité en B événementiel,” *Technique et Science Informatiques*, vol. 25, no. 1, pp. 73–102, 2006.

[16] M. Leuschel and M. Butler, “ProB: An Automated Analysis Toolset for the B Method,” *International Journal on Software Tools for Technology Transfer*, 2008.

[17] D. Plagge and M. Leuschel, “Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more,” *International Journal on Software Tools for Technology Transfer*, 2008.

[18] M. Samia, H. Wiegard, J. Bendisposto, and M. Leuschel, “High-Level versus Low-Level Specifications: Comparing B with Promela and ProB with Spin,” in *Proceedings TFM-B 2009*, Attiogbe and Mery, Eds. APCB, June 2009.

[19] “bcomp, the B Compiler,” http://sourceforge.net/projects/bcomp/.

[20] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, “Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems,” in *Proc. of Workshop on Verification and Control of Hybrid Systems III*, ser. Lecture Notes in Computer Science, no. 1066. Springer–Verlag, Oct. 1995, pp. 232–243.