Proving the absence property pattern using the B method

Amel Mammar, Marc Frappier, Raphael Chane-Yack-Fa

Proving the Absence Property Pattern Using the B Method

Amel Mammar
Institut Telecom SudParis
CNRS/SAMOVAR
Paris, France
amel.mammar@it-sudparis.eu

Marc Frappier
Université de Sherbrooke
GRIL
Sherbrooke (Québec), Canada
marc.frappier@usherbrooke.ca

Raphael Chane-Yack-Fa
Université de Sherbrooke
GRIL
Sherbrooke (Québec), Canada
Raphael.Chane-Yack-Fa@USherbrooke.ca

***Abstract*—Dynamic properties are very useful in the specification of Information Systems (IS) and security policies. They allow the user to express properties that involve several states of a system. Indeed, invariance properties do not permit to cover such kind of properties. In this paper, we suggest a formal approach, based on the use of the B method, to verifying absence properties of the form $\mathsf{Abs}(P_2,\ \mathsf{From}\ P_1\ \mathsf{Until}\ P_3)$ that express that some states, represented by predicate P_2, should not be reached starting from a state that satisfies P_1 until a state satisfies P_3 is reached. Our proposal consists in defining two proof obligations based on weakest preconditions that are sufficient and necessary to prove that a system verifies such a property.**

***Keywords*-Verification; Temporal properties; Absence patterns; B Method.**

I. Introduction

The specification and the verification of dynamic properties play an essential role in the development process of Information Systems (IS). Contrary to invariance properties, dynamic properties permit to describe advanced properties that depend on several states occurring at different moments (*i.e.*, temporal properties). In this paper, we are particularly interested in the dynamic properties that can be expressed by the absence pattern introduced in [10]: $\mathsf{Abs}(P_2,\ \mathsf{From}\ P_1\ \mathsf{Until}\ P_3)$. This pattern expresses that some states, represented by predicate P_2, should not be reached if the system has been in a state that satisfies P_1 until predicate P_3 becomes fulfilled. In practice, this kind of properties is very common and useful in several domains and applications. In a ticket sale system for instance, we should verify that after reserving a ticket, the client does not get it before performing the payment. Similarly in the transport domain, a signal should remain closed after a train has passed it until the route becomes completely free.

Introduced by J.R Abrial [1], B is a formal method for developing safe systems. A *safe system* satisfies some safety properties and does no harm. To this aim, a B developer has to express such properties as invariants and specify the adequate conditions under which operations should be executed in order to maintain the desired properties. These conditions, called preconditions, aim at reducing the set of allowed system behaviors to those that preserve the invariants. In B, the temporal (dynamic) properties are not supported. Ad hoc techniques can be used to encode a dynamic property into invariants, but they require tweaking of the specification, by adding new state variables, thus making the specification more complex.

In this paper, we propose a formal approach to verifying dynamic properties, expressed by the absence pattern $\mathsf{Abs}(P_2,\ \mathsf{From}\ P_1\ \mathsf{Until}\ P_3)$), using the B formal method. Our approach consists in defining sufficient and necessary conditions that ensure the satisfaction of such properties.

II. The B method and case study presentations

A. Overview of B

In B, the specifications are organized into abstract machines. Each machine encapsulates state variables on which operations are expressed. The set of the possible states of the system are described using an invariant. The invariant is a predicate in a simplified version of the ZF-set theory [9], enriched with many relational operators. Operations are specified in the Generalized Substitution Language (GSL) [1]. A substitution is like an assignment statement. An elementary substitution is denoted by $x := E$, where x is a state variable and E an expression. It allows one to identify which variables are modified by the operation, while avoiding mentioning ones which are not. The generalization allows the definition of non-deterministic and preconditioned substitutions. To ensure the correctness of a B specification, a set of proof obligations is generated for each B component. These proofs aim at verifying that the invariant of the system is satisfied after the execution of each operation. Of course, such an invariant is assumed to be satisfied before an operation is executed. For each invariant Inv and operation op whose precondition and substitution are P and S respectively, the following proof obligation is raised: $(Inv \wedge P) \Rightarrow [S]Inv$. More explanations about the B notation will be given when needed.

B. Case study presentation

We illustrate our proposal with a management system that deals with ticket sales to customers for some destinations. We make the assumption that the number of places on each flight is equal to $NbPlaces$. If a place is available on the desired flight, the customer gets his/her ticket (**GetTicket**) otherwise, he/she is put in the waiting queue associated with the flight (**WaitQueue**). Such a customer will get a ticket when a place becomes free on the flight and he/she is at the head of the waiting queue (**TakeTicket**). The B specification corresponding to this system is depicted in Figure 1 where the following operators are used.

- $x \mapsto y$ denotes the pair (x, y).
- The domain of a relation r is defined as $dom(r) = \{x \mid \exists\, y \cdot x \mapsto y \in r\}$
- the negative domain restriction of relation r by set X is defined as $X \lhd\!\!\!- r = \{x \mapsto y \mid x \mapsto y \in r \wedge x \notin X\}$.
- the override of relation r_1 by relation r_2 is defined as $r_1 \lhd\!\!\!+ r_2 = (dom(r_2) \lhd\!\!\!- r_1) \cup r_2$.
- A sequence of length n of elements of type X is represented in B with a total function of type $1..n \rightarrow X$.
- The set $\mathbf{iseq}(X)$ denotes the injective sequences of elements of X.
- $s \leftarrow x$ denotes the insertion of element x at the end of sequence s.
- $tail(s)$ represents sequence s, without its first element.
- $first(s)$ represents the first element of sequence s.
- The substitution $S_1 \parallel S_2$ denotes the simultaneous execution of S_1 and S_2, assuming that S_1 and S_2 operate on disjoint sets of modified variables.
- Given an operation op of the form **PRE** P **THEN** T **END**, we let S_{op} denote the substitution T of op and $pre(op)$ its preconditions .

Using the prover of AtelierB, we have proved the correctness of the $FlightSystem$ specification by generating 12 proof obligations in order to ensure that the execution of each operation re-establishes the invariant: 10 of them have been discharged automatically while the others have required our intervention to help the prover find the right rules to apply. Nevertheless, such proof obligations do not guarantee fairness to ensure, for instance, that if a customer cu_1 is put in a waiting queue of a flight fl_1 before a customer cu_2, then he/she will get a place before cu_2. This property can be expressed by:

$$\begin{aligned} &\mathsf{Abs}(cu_2 \mapsto fl_1 \in tickets, \\ &\quad \mathsf{From}(cu_1 \in ran(waitingQueue(fl_1)) \wedge \\ &\qquad cu_2 \notin ran(waitingQueue(fl_1)) \wedge \\ &\qquad cu_2 \mapsto fl_1 \notin tickets) \\ &\quad \mathsf{Until}(cu_1 \mapsto fl_1 \in tickets)) \end{aligned} \tag{1}$$

The rest of the paper addresses the proof of such dynamic properties by defining the B proof obligations that are necessary and sufficient to prove them.

Machine
  $FlightSystem$
Sets
  Customers; Flights
Variables
  tickets, waitingQueue
Constants
  $NbPlaces$
Properties
  $NbPlaces \in NAT_1$
Invariant
  $tickets \in Customers \leftrightarrow Flights \wedge$
  $waitingQueue \in Flights \rightarrow \mathbf{iseq}(Customers) \wedge$
  $\forall fl.\ (fl \in Flights \Rightarrow \mathbf{card}(tickets^{-1}[\{fl\}]) \leq NbPlaces)$
DEFINITIONS
/*$Index(fl, cu)$ gives the rank of a customer cu
    in the waiting queue of a flight fl*/
  Index(fl,cu) $\widehat{=} (waitingQueue(fl))^{-1}(cu)$
Operations
GetTicket(cu , fl)$\widehat{=}$
PRE
  $cu \in Customers \wedge fl \in Flights \wedge$
  $\mathbf{card}(tickets^{-1}[\{fl\}]) < NbPlaces \wedge$
  $waitingQueue(fl) = []$
THEN
  $tickets := tickets \cup \{cu \mapsto fl\}$
END;
TakeTicket(cu , fl)$\widehat{=}$
PRE
  $cu \in Customers \wedge fl \in Flights \wedge$
  $\mathbf{card}(tickets^{-1}[\{fl\}]) < NbPlaces \wedge$
  $\mathbf{first}(waitingQueue(fl)) = cu$
THEN
  $tickets := tickets \cup \{\mathbf{first}(waitingQueue(fl)) \mapsto fl\} \parallel$
  $waitingQueue := waitingQueue \lhd\!\!\!+$
    $\{fl \mapsto \mathbf{tail}\ (waitingQueue(fl))\}$
END;
WaitQueue(cu, fl)$\widehat{=}$
PRE
  $cu \in Customers \wedge fl \in Flights \wedge$
  $cu \notin ran(waitingQueue(fl)) \wedge cu \mapsto fl \notin tickets \wedge$
  $(\mathbf{card}(tickets^{-1}[\{fl\}]) = NbPlaces \vee$
    $(waitingQueue(fl) \neq []))$
THEN
  $waitingQueue := waitingQueue \lhd\!\!\!+$
    $\{fl \mapsto ((waitingQueue(fl) \leftarrow cu)\}$
END
END

Figure 1. The B specification of the tickets management system

III. Proving the Absence Patterns

A. Derivation of the necessary and sufficient conditions

In this section, we show the derivation of the B sufficient and necessary assertions to prove the absence pattern $\mathsf{Abs}(P_2,\ \mathsf{From}\ P_1\ \mathsf{Until}\ P_3)$ and its application to the running case study. Our proposal consists in demonstrating that starting from a state satisfying P_1, the system will behave as follows (See Figure 2):

1) In the state that satisfies P_1:
   - Predicate P_3 is satisfied: the property is fulfilled and the verification stops, or
   - Predicates P_2 and P_3 are not satisfied: the property is not violated yet. The verification process must continue because neither P_2 nor P_3 is true.

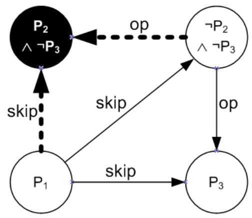

Figure 2. Graphical representation of property $\mathsf{Abs}(P_2, \mathsf{From}\ P_1\ \mathsf{Until}\ P_3)$

- otherwise, the property is violated. That case is represented by dashed lines and a black state in Figure 2 and denotes the forbidden behavior.

These cases are depicted by a transition labelled with the $skip$ action that does nothing.

2) Being in state $(\neg P_2 \wedge \neg P_3)$, we have to verify that the execution of any operation op makes the system move to state P_3 or stay in state $(\neg P_2 \wedge \neg P_3)$.

This yields the following proof obligations:

1) the temporal property is satisfied in the state where P_1 holds:

$$\forall(\overrightarrow{x}, \overrightarrow{y}).(P_1 \Rightarrow (\neg P_2 \vee P_3)) \tag{2}$$

2) predicate P_2 should stay not satisfied while P_3 is not satisfied yet

$$\forall(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{v}).(\neg P_2 \wedge pre(op) \Rightarrow [S_{op}](\neg P_2 \vee P_3)) \tag{3}$$

where $\overrightarrow{x}$ denote the values of the machine variables $(x_1, \ldots, x_n)$, $\overrightarrow{y}$ are the variables $(y_1, \ldots, y_m)$ that may appear in predicates P_1, P_2 and P_3 and which are distinct from variables $\overrightarrow{x}$, and $\overrightarrow{v}$ denote the parameters of operation op.

Let us stress that (3) should be satisfied only on intermediate states between P_1 and P_3. However predicate $\neg P_2$ may be larger than the set of these intermediate states, thus we may have to restrict $\neg P_2$ (*i.e.*, enlarge P_2) in order to be exactly equal to this set. In order to be clearer, let us illustrate that on the running case study and try to prove (3) for property (1) and operation **TakeTicket**:

$$\forall(tickets, waintingQueue, cu_1, cu_2, fl_1, cu, fl).$$
$$\left(\begin{array}{c} cu_2 \mapsto fl_1 \notin tickets \wedge pre(TakeTicket) \\ \Rightarrow \\ [S_{TakeTicket}]\,(cu_2 \mapsto fl_1 \notin tickets \vee cu_1 \mapsto fl_1 \in tickets) \end{array}\right)$$

Let us remark that the set of states denoted by predicate $(cu_2 \mapsto fl_1 \notin tickets)$ includes states such that a place is available on flight fl_1 and customer cu_2 is at the head of the waiting queue, i.e, before customer cu_1. It is obvious that such states violate the previous proof obligation since it is possible to execute operation **TakeTicket** and make a reservation for customer cu_2 $(cu = cu_2, fl = fl_1)$. These counterexamples are found using a model checker like ProB [17] or Alloy [7]. Nevertheless, such a counterexample is a false one since we know that such states do not belong to $From\_To(P_1, P_3)$. Indeed, position of customer cu_2 cannot be before that of customer cu_1 in the waiting queue, since new waiting customers are added at the end of the queue. In addition, cu_1 remains in the queue until he gets a place. So, the specifier, given his knowledge of the specification and the counter-example found, has to enrich predicate P_2 in order to rule out this false counterexample. So now, we have to enlarge P_2 with P' defined by:

$$\left(\begin{array}{c} cu_1 \notin ran(waitingQueue(fl_1)) \\ \vee \\ \left(\begin{array}{c} cu_2 \in ran(waitingQueue(fl_1)) \\ \wedge \\ Index(fl_1, cu_2) < Index(fl_1, cu_1) \end{array}\right) \end{array}\right)$$

We have to repeat the process until no counterexample is found. By doing that, we will characterize all the states belonging to $From\_To(P_1, P_3)$. This leads to the following theorem.

Theorem 1: Let P_1, P_2 and P_3 be three predicates. Property $(\mathsf{Abs}(P_2, \mathsf{From}\ P_1\ \mathsf{Until}\ P_3))$ is satisfied iff there exists a predicate P' such that the following proof obligations hold for each operation op:

(i) $\forall(\overrightarrow{x}, \overrightarrow{y}).(P_1 \Rightarrow (\neg(P_2 \vee P') \vee P_3))$

(ii) $\forall(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{v}).(\neg(P_2 \vee P') \wedge pre(op) \Rightarrow [S_{op}](\neg(P_2 \vee P') \vee P_3))$ ■

B. Proving the Assertions in B

In this section, we report the results obtained on our case study and the absence property (1). Applying the proof rules (i) and (ii), provided in Theorem 1 gives the following proof obligations (POs):

- **PO1.** $\forall \overrightarrow{v}.(P_1 \Rightarrow (\neg(P_2 \vee P') \vee P_3))$
- **PO2.** $\forall \overrightarrow{v}.(\neg(P_2 \vee P') \wedge pre(op) \Rightarrow [S_{op}](\neg(P_2 \vee P') \vee P_3))$

where $\overrightarrow{v}$ includes the free variables of the absence property $(\{cu_1, cu_2, fl_1\})$ and the formal input parameters of operation **op**. Predicates P_1, P_2, P' and P_3 are as follows:

$$\begin{array}{rcl} P_1 & = & \left(\begin{array}{c} cu_1 \in ran(waitingQueue(fl_1)) \\ \wedge \\ cu_2 \notin ran(waitingQueue(fl_1)) \\ \wedge \\ cu_2 \mapsto fl_1 \notin tickets \end{array}\right) \\ P_2 & = & (cu_2 \mapsto fl_1 \in tickets) \\ P' & = & \left(\begin{array}{c} cu_1 \notin ran(waitingQueue(fl_1)) \\ \vee \\ \left(\begin{array}{c} cu_2 \in ran(waitingQueue(fl_1)) \\ \wedge \\ Index(fl_1, cu_2) < Index(fl_1, cu_1) \end{array}\right) \end{array}\right) \\ P_3 & = & (cu_1 \mapsto fl_1 \in tickets) \end{array}$$

Proof obligation	Automatic Proofs	Interactive Proofs
PO1	1	0
PO2	1	6

Table I
PROOF RESULTS

To be discharged using the prover of AtelierB, proof obligations (**PO1**) and (**PO2**) are added as assertions (clause **ASSERTIONS** of the B notation) to machine *FlightSystem* of page 2. Table I gives the statistics we obtained on operations **GetTicket**, **TakeTicket** and **WaitQueue**. The proof are not very difficult, the automatic prover fails to discharge them because they require several steps and also the following rule, related to the sequence structure, is missing in its rule base:

$$a \in \mathbf{iseq}(b)) \wedge$$
$$a \neq [] \wedge$$
$$c \in \mathbf{ran}(\mathbf{tail}(a))$$
$$\Rightarrow$$
$$(\mathbf{tail}(a))^{-1}(c) = a^{-1}(c) - 1$$

IV. Conclusion

In this paper, we have defined necessary and sufficient conditions for proving dynamic properties of the form $\mathsf{Abs}(P_2,\ \mathsf{From}\ P_1\ \mathsf{Until}\ P_3)$. Such a property ensures that starting from a state verifying P_1, the system will not reach a state satisfying P_2 before predicate P_3 becomes true. The key idea of the suggested approach is to characterize the set of states that can be reached starting from any state verifying P_1 and before reaching any state that would satisfy P_3. This set being defined, the proposal consists in proving that the execution of any operation on these states makes the system move to a state verifying P_3 or $\neg P_2$. To ensure the correctness of the approach, proofs are carried out to formally establish the soundness and the completeness of the defined conditions.

Future work include the automation of this approach to make it more workable. We also plan to extend our approach to take into account other kinds of property patterns that would be interesting in information systems. An example of these patterns is the Response pattern that permits to specify that a state/event is always followed by another state/event. Such a pattern will be used to state, for instance, that a customer will get a place if he/she requests it.

References

[1] J.R. Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.

[2] http://www.atelierb.eu

[3] M.E. Beato and M. Barrio-Solorzano and C. Cuesta and P. De La Fuente, *UML Automatic Verification Tool with Formal Methods*, volume 127, number 4, Electronic Notes in Theoretical Computer Science, 2005.

[4] T. de Champs, B. Abdulrazak, H. Pigot, M. Ouenzar, M. Frappier, B. Fraikin, *Pervasive Safety Application with Model Checking in Smart Houses: the INOVUS Intelligent Oven*, Workshop on Smart Environments to Enhance Health Care, in 2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM), IEEE Computer Society, pp 586-591.

[5] A.E. Emerson and J.Y. Halpern, *Decision Procedures and Expressiveness in the Temporal Logic of Branching Time*, J. Comput. Syst. Sci., volume 30, number 1, 1985.

[6] M. Frappier and B. Fraikin and R. Chane-Yack-Fa and M. Ouenzar, *Comparison of Model Checking Tools for Information Systems*. In J-S. Dong and H. Zhu, editors, ICFEM, volume 6447 of LNCS, pages 581596, Springer, 2010.

[7] D. Jackson, Software Abstractions, Logic, Language, and Analysis, MIT Press, 2012.

[8] Y. Kesten and Z. Manna and A. Pnueli, *Verifying Clocked Transition Systems*. In R. Alur, T-A. Henzinger, and ED. Sontag, editors, Hybrid Systems, volume 1066 of LNCS, pages 1340, Springer, 1995.

[9] K. Kunen, *Set theory : An Introduction to Independence Proofs, Studies in logic and the foundations of mathematics*, volume 102 of Elsevier North-Holland, 1980.

[10] M.B. Dwyer and G.S. Avrunin and J.C. Corbett, *Patterns in Property Specifications for Finite-State Verification*, in Proceedings of the 21st International Conference on Software Engineering, 1999

[11] M. Leuschel and D. Plagge, *Seven at one Stroke: LTL Model Checking for High-Level Specifications in B, Z, CSP, and more*. Technical report, 2007

[12] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*, Springer-Verlag New York, Inc., 1995

[13] K. L. McMillan, *Symbolic Model Checking. An Approach to the State Explosion Problem*. Carnegie Mellon University. CMU-CS-92-131, 1992

[14] C.C. Morgan, *Programming from Specifications*, Prentice Hall, 1998

[15] OMG, *Object Management Group (OMG): Unified Modeling Language (UML 2.0).* http://www.uml.org/

[16] A. Pnueli, *The Temporal Logic of Programs*, in 18^{th} Annual Symposium on Foundations of Computer Science(FOCS), 1977

[17] ProB, http://www.stups.uni-duesseldorf.de/ProB

[18] J. Simmonds and M. Chechik and S. Nejati and E. Litani and B. O'Farrell, *Property Patterns for Runtime Monitoring of Web Service Conversations*, In 8th International Workshop on Runtime Verification, Martin Leucker ed., volume 5289, Lecture Notes in Computer Science, Springer, 2008