Verifying the Absence Property Pattern

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

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

***Abstract*—Temporal properties are very common in various classes of systems, including information systems and security policies. This paper investigates two verification methods, proof and model checking, for one of the most frequent patterns of temporal property, the absence pattern. We explore two model-based specification techniques, B and Alloy, because of their adequacy for easily specifying systems with complex data structures, like information systems. We propose a first-order, assertion-based, sound and complete strategy to verify the absence pattern. This enables the proof of the absence pattern using conventional first-order provers. It also significantly increases the size of the models that can be checked, when compared to traditional LTL model checking techniques. The approach is illustrated throughout a case study.**

***Keywords*-Verification; Temporal properties; Absence properties; B Method; Alloy; Proofs; Model Checking.**

I. Introduction

The specification and the verification of temporal (also called 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 on traces of systems. In this paper, we are particularly interested in the dynamic properties that can be expressed by the absence pattern introduced in [5]: $\mathsf{Abs}(P_2,\ \mathsf{After}\ P_1\ \mathsf{Until}\ P_3)$. This pattern expresses that some states, represented by predicate P_2, should not be reached after the system entering a state that verifies P_1 until predicate P_3 becomes fulfilled; P_3 does not need to be reached. The absence property can be expressed in LTL [19] as $\Box(P_1 \Rightarrow \mathsf{X}(\neg P_2 \mathsf{W} P_3))$.

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 the ticket before paying it. Similarly in the transport domain, a signal should remain closed after a train has passed it until the route becomes completely free. In a typical security policy, some actions are forbidden until proper authorization has been granted.

There are three main approaches to verifying temporal properties: testing, model checking or theorem proving. Testing is the most widely used method in practice, but it suffers from a lack of automation and very limited coverage of the test space. Model checking has the advantage of being more automatic, but it quickly suffers from combinatorial explosion, thus limiting the size of the models checked. Consequently, the confidence in the correctness of the system is limited to the relatively small size of the models analyzed. Theorem proving requires more human intervention and sometimes considerable expertise, but it certifies the correctness of the system, since proofs are valid for any models. However, little attention has been paid so far to the proof of temporal properties, because they are more complex to carry out than typical invariance properties. In this paper, we focus on proving and model checking as two alternatives, because of their ability to ensure a higher level of correctness than testing. These two verification techniques can be applied either on the source code or on an abstract specification of the system. Verifying the source code is more desirable, because it ensures correctness of the actual system. However, verifying the source code is harder than verifying an abstract specification, because too many implementation details are involved. Model checking is often inapplicable to source code without constructing an abstraction of the source code, thereby decreasing the level of automation and potentially decreasing the level of confidence in the correctness of the system [21]. Proofs on source code are also more difficult to carry out, because there are considerably more proof obligations to deal with.

The advent of service-oriented architectures, generative programming techniques, domain specific languages, and automatic refinement of abstract specifications into executable source code, foster the use of abstract specifications. This opens new perspectives for verifying properties on abstract specifications rather than source code. In this paper, we explore this new perspective for the absence pattern. We have chosen two model-based methods to specify and verify specifications: B [1] and Alloy [9]. Their specification languages are rich enough to easily deal with abstract specification of various classes of systems. They are supported by effective tools for either proof or model checking.

The B method has been successfully used over the last two decades on numerous industrial projects of significant size (up to 158 KSLOC) for constructing safety critical systems in various domains like rail transport, aeronautics, automobile, defence and R&D. More than 25 cities all over the world now use subway and train systems whose control systems were developed and proved correct using B. This method supports the whole system development life cycle, from specification to implementation, using refinement, and supports formal proof of correctness of the implementation with respect to the abstract specification. Invariance properties can be expressed and proved. 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, making the specification more complex. Thus, they remain complex to achieve in practice. B is supported by a suite of tools. We use Atelier B [2] for proofs and ProB [13] for model checking. ProB supports LTL and a subset of CTL. We could also use the new derivative of B, Event B, which offers an event-driven view of the system and a different refinement mechanism. Event B is also supported by tools like Rodin and ProB. For our purpose, the differences between the two notations are irrelevant.

Alloy is a language based on first-order logic with integers and relations. The Alloy analyzer translates first-order formulas into boolean formulas and uses conventional SAT solvers like SAT4J, MiniSat, and ZChaff, to find models of specifications and to check first-order logic assertions on all models. The relational language of Alloy is roughly as expressive as the the set-theoretic language of the B notation, thus the same classes of systems can be specified using either languages. Alloy does not support the verification of temporal formulas expressed in LTL or CTL. To do so, one has to translate the temporal formula into equivalent first-order formulas. This paper provides an example of such a transformation for the absence pattern.

In [7], six models checkers are compared for data-intensive specifications like information systems. ProB and Alloy emerged as two of the most versatile model-checking tools for data-intensive systems. Alloy was the most efficient tool in terms of model size and verification time. ProB was the easiest to use for specification and verification, because of its rich data structures and its support of both LTL and CTL. Moreover, ProB allows universal quantification over state variables in LTL formula, something which is not supported in traditional state-based checkers like SMV or SPIN.

The contributions of this papers are as follows. It proposes an assertion-based technique for verifying the absence pattern $\mathsf{Abs}(P_2,\ \mathsf{After}\ P_1\ \mathsf{Until}\ P_3)$, which can be formalized in LTL as $\Box(P_1 \Rightarrow \mathsf{X}(\neg P_2 \mathsf{W} P_3))$. The assertions are first-order formula; they do not contain any temporal operators. This technique is inspired from the work of Pnueli and Manna [14], which provides a proof system for some LTL formulas. We propose three different proof rules for generating these assertions, each of them being more advantageous in specific cases. These rules are equivalent, sound and complete. These assertions can be either proved using a first-order theorem prover like the Atelier B prover, or model checked.

These assertions being first-order formula, instead of temporal formula, they can be rapidly checked using Alloy on larger models than the corresponding LTL formula in conventional LTL model checkers like ProB, SPIN, FDR2, CADP and SPIN. Because larger models can be checked, our approach ensures a higher degree of confidence in the correctness of the specification when model checking is used. To illustrate our approach, we use the same case study as in [7], a library system, to allow for comparisons. In [7], the largest model that could be checked for the given temporal formulas contained 5 books and 5 members using SPIN. FDR2, CADP, NuSMV and ProB were limited to 3 members and 3 books without blowing up memory on a 4GB workstation. SPIN performs slightly better due to its on-the-fly checking strategy. Using our assertion-based approach, Alloy can check the absence pattern for 5 members and 5 books in less than 0.6 seconds, and up to 36 members and 36 books in less than 9 minutes, with less than 665MB of memory.

Our assertion-based approach also enables provers in model-based techniques like B, Z, VDM and ASM to prove absence properties, without having to tweak a specification by adding new variables. The drawback of our approach is that it requires one to find a formula similar to an invariant, like the loop invariant used in the correctness proof of a loop. However, we propose heuristics to find this invariant-like formula and use the ProB model checker or Alloy to guess this invariant using counterexamples generated by the model-checking process.

In the literature, the common approaches to verify patterns of dynamic properties are mainly based on the use of model checkers and testing techniques. In [3], a verification tool that permits to check a wide range of properties on UML dynamic diagrams [18] is provided. These properties are expressed according to the patterns defined in [5]. To do that, the UML diagrams together with the properties to verify are translated and checked with SMV [16] to detect any property violation. These same patterns are verified in [22] using a monitoring based-approach to guarantee the correctness of web service conversations modeled with UML2 sequence diagrams. Global sequence diagrams, representing the web service conversations and the properties, are mapped into non-deterministic finite automata in order to check them.

In [14], proof rules are provided for several forms of LTL formula. The rule WAIT is the closest one that could be used to prove the absence property, by applying it to each operation *op* of a B machine. The B notation "$[op]P$" denotes Dijkstra's weakest-precondition $wp(op, P)$ of operation op wrt predicate P, *i.e.*, the states where op is guaranteed to terminate in P. $Pre(op)$ denotes the precondition of operation op.

$$\frac{P_1 \Rightarrow P_3 \vee \phi, \quad \phi \Rightarrow \neg P_2, \quad Pre(op) \wedge \phi \Rightarrow [op](P_3 \vee \phi)}{P_1 \Rightarrow (\neg P_2 \mathsf{W} P_3)}$$

Because of the next operator involved in the LTL expression of the absence pattern, this rule is close, but inappropriate. In fact, when $P_1 \Rightarrow P_3$ and if P_2 holds after P_1 and P_3 does not, the absence property $\mathsf{Abs}(P_2,\ \mathsf{After}\ P_1\ \mathsf{Until}\ P_3)$ is falsified while the WAIT rule satisfies it. In addition, the proof of premise $(\phi \Rightarrow \neg P_2)$ may require invariants, which must themselves be proved over all operations of the system, thus making it quite hard to use in practice. We propose rules to avoid this specific problem by defining necessary and sufficient premises whose proof only uses the invariant of the original specification (See section V).

This paper is structured as follows. Section II presents a quick overview of the B method, while Section III describes the case study, a subset of the library specification used in [7]. Section IV describes the assertions that can be used to prove the absence property. Then we show how these assertions can be proved in B and model-checked in Alloy in Section V. We propose two additional variants of these assertions in Section VI. We describe how the soundness and completeness of these assertions can be verified in Alloy. A formal hand-made proof is provided in the appendix of this paper. Finally, we conclude with an appraisal of our contributions in Section VIII.

II. Overview of the B method

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. We define $Pre(op)$ as $[op]\mathsf{true}$.

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 [12], 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: $(Inv \wedge P) \Rightarrow [S]Inv$.

III. Case study presentation

We illustrate our proposal with a library management system. The system has to manage book loans to members. Each member may either borrow (**Lend**) or reserve (**Reserve**) a book if the latter is already lent to another member. Several reservations may be made on the same book using a waiting queue. After doing a reservation on a book, a member borrows (**Take**) the book when the members, who made reservations before her/him, have already borrowed and returned (**Return**) the book. Also, we make the assumption that a member cannot borrow more than $MaxNbLoans$ books at the same time. A subset of the B specification corresponding to this system is included in figure 1, where the following operators are used.

- $x \mapsto y$ denotes the pair (x, y).
- 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 $r_1 \lhd\!\!+ r_2 = (dom(r_2) \lhd\!\!\!- r_1) \cup r_2$.
- The domain of a relation r is defined as $dom(r) = \{x \mid \exists\, y \cdot x \mapsto y \in r\}$
- A sequence of length n of elements of 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 of the form **PRE** P **THEN** T **END**, we let S_{op} denote the substitution T of op.

Using the prover of Atelier B, we have proved the correctness of the $Library$ specification. Atelier B generates 12 proof obligations to ensure that the execution of each operation preserves the invariant: 6 of them are discharged automatically by the prover while the others require 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 member me_1 reserves a book bo_1 before a member me_2, he will get the book before me_2. Such a property is expressed using the absence pattern introduced in [5] as follows:

Machine *Library*
Sets *Books; Members*
Variables *loan, reservation*
Constants $MaxNbLoans$
Properties $MaxNbLoans \in NAT_1$
Invariant
$loan \in Books \nrightarrow Members \wedge$
$reservation \in Books \rightarrow \mathbf{iseq}(Members) \wedge$
$\forall me. (me \in Members \Rightarrow$
$\mathbf{card}(loan^{-1}[\{me\}]) \leq MaxNbLoans)$
DEFINITIONS
/*$Index(bo, me)$ gives the position of a member me in the reservation queue of a book bo*/
Index(bo,me) $\widehat{=} (reservation(bo))^{-1}(me)$
Operations
Lend(me , bo)$\widehat{=}$
PRE $me \in Members \wedge bo \in Books \wedge$
$bo \notin dom(loan) \wedge reservation(bo) = [] \wedge$
$card(loan^{-1}[\{me\}]) < MaxNbLoans$
THEN
$loan := loan \mathbin{<\!\!+} \{bo \mapsto me\}$
END;
Take(me , bo)$\widehat{=}$
PRE $me \in Members \wedge bo \in Books \wedge$
$bo \notin dom(loan) \wedge \mathbf{first}(reservation(bo)) = me) \wedge$
$card(loan^{-1}[\{me\}]) < MaxNbLoans$
THEN
$loan := loan \mathbin{<\!\!+} \{bo \mapsto me\} \|$
$reservation := reservation \mathbin{<\!\!+}$
$\{bo \mapsto tail(reservation(bo))\}$
END;
Reserve(me, bo)$\widehat{=}$
PRE $me \in Members \wedge bo \in Books \wedge$
$me \notin ran(reservation(bo)) \wedge bo \mapsto me \notin loan \wedge$
$((bo \in dom(loan)) \vee (reservation(bo) \neq []))$
THEN
$reservation := reservation \mathbin{<\!\!+}$
$\{bo \mapsto ((reservation(bo) \leftarrow me)\}$
END;
Return(bo)$\widehat{=}$
PRE $bo \in Books \wedge bo \in dom(loan)$ **THEN**
$loan := \{bo\} \mathbin{\lhd\!\!\!-} loan$
END
END

Figure 1. The B specification of the library system

$$\begin{aligned}
&\mathsf{Abs}(bo_1 \mapsto me_2 \in loan,\\
&\quad \mathsf{After}(me_1 \in ran(reservation(bo_1)) \wedge\\
&\qquad me_2 \notin ran(reservation(bo_1)) \wedge\\
&\qquad bo_1 \mapsto me_2 \notin loan)\\
&\quad \mathsf{Until}(bo_1 \mapsto me_1 \in loan))
\end{aligned} \tag{1}$$

which is expressed in LTL by:

$$\begin{aligned}
&\Box(me_1 \in ran(reservation(bo_1)) \wedge\\
&\quad me_2 \notin ran(reservation(bo_1)) \wedge\\
&\quad bo_1 \mapsto me_2 \notin loan\\
&\quad \Rightarrow\\
&\quad \mathsf{X}(bo_1 \mapsto me_2 \notin loan \;\mathsf{W}\; bo_1 \mapsto me_1 \in loan))
\end{aligned}$$

We have checked this LTL formula with ProB on the specification of figure 1, which does not include all the operations used in the specification of [7] (five operations are missing, *i.e.*, the operations for creating and deleting books and members, plus the cancellation of a reservation). Thus, its transition system is smaller than that of the complete library system [7]. On this reduced state space, the largest model that ProB can check contains 4 books and 6 members. As indicated before, on the complete specification, ProB is limited to 3 books and 3 members. This is why the rest of the paper addresses the proof of such properties by defining the B proof obligations that are necessary and sufficient to prove them.

IV. Defining Assertions to Verify the Absence Pattern

In this section, we describe the process we have defined to verify that a system satisfies an absence property $\mathsf{Abs}(P_2, \mathsf{After}\ P_1\ \mathsf{Until}\ P_3)$ and we use the case study of the previous section to illustrate it.

A. Formal Definition of the Absence Pattern

Because we use an assertion-based method, our definition of the absence pattern is slightly stricter than the temporal formula $\Box(P_1 \Rightarrow \mathsf{X}(\neg P_2 \mathsf{W} P_3))$. This formula is defined over traces of a system; hence, it applies only to states reachable from the initial state. Our approach will consider all states reached from P_1; however, some of the states of P_1 may not be reachable from the initial state. Let σ be a state satisfying P_1, but not reachable from the initial state. Furthermore, suppose that a transition from σ leads to a state satisfying P_2. Since σ is not reachable from the initial state, it does not falsify the temporal property. However, it will falsify our definition of the absence property, since we consider all states in P_1. In that particular (and unfrequent) case, one can still use our assertion-based approach by restricting predicate P_1 to states reachable from the initial state. Proofs and model checking are easier to conduct when our stricter definition is used, without sacrificing generality.

To formally define the semantics of the absence pattern: let us introduce the following notations.

1) Let Σ be the set of states of machine M. Let $\sigma \models P$ denote that state σ satisfies predicate P. Let $op(\sigma)$ denote the states reached after executing operation op from state σ (if the operation is nondeterministic, several states are possible).
2) For a given predicate P, Σ_P denotes the subset of Σ that satisfies P:
$$\Sigma_P = \{\sigma \mid \sigma \in \Sigma \wedge\ \sigma \models P\}$$
3) For a given predicate P, $Paths(P)$ denotes the set of the paths starting from P by executing the different operations Ops of the system:
$$Paths(P) = \{(\sigma_0, \sigma_1, \ldots) \mid \sigma_0 \in \Sigma_P \wedge \forall i.(i \geq 1 \Rightarrow \exists\, op.(\sigma_i \in op(\sigma_{i-1})))\}$$

4) For a given $path = (\sigma_0, \sigma_1, \ldots, \sigma_i, \ldots)$ and an i such that $(0 \le i)$, $path[i]$ denotes the state σ_i in $path$.
5) For a path $path = (\sigma_0, \sigma_1, \ldots, \sigma_n)$, $path[i..j]$ denotes the sub-path $(\sigma_1, \ldots, \sigma_j)$.
6) For two predicates P_1 and P_2, $From\_To(P_1, P_2)$ denotes the sub-paths that start from a successor of a P_1 state and do not contain any state that satisfies P_2:

$$\begin{aligned} From\_To(P_1,P_2) = \{path[1..j] \mid path \in Paths(P_1) \wedge \\ j \ge 1 \wedge \forall k.(1 \le k \le j \Rightarrow path[k] \models \neg P_2)\} \end{aligned}$$

We say that a machine M satisfies property

$$\mathsf{Abs}(P_2, \mathsf{After}\ P_1\ \mathsf{Until}\ P_3)) \tag{2}$$

iff:

$$\forall path. \left(\begin{array}{c} path \in Paths(P_1) \\ \Rightarrow \\ \forall i. \left(\begin{array}{c} \left(\begin{array}{c} i > 0 \\ \wedge \\ path[i] \models P_2 \end{array} \right) \\ \Rightarrow \\ \exists j. \left(\begin{array}{c} j > 0 \wedge j \le i \\ \wedge \\ path[j] \models P_3 \end{array} \right) \end{array} \right) \end{array} \right) \tag{3}$$

In the next subsection, we present our approach to defining the B proof obligations that permit to prove an absence property.

B. Definition of the Proof Obligations

To prove property (3) on a B machine, our proposal consists in demonstrating that starting from a state σ satisfying P_1, the system will behave as follows (See Figure 2):

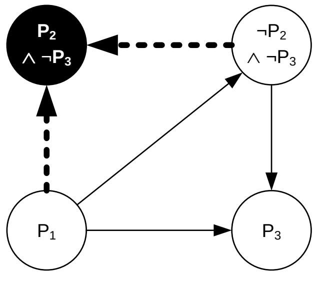

Figure 2. Graphical representation of property $\mathsf{Abs}(P_2, \mathsf{After}\ P_1\ \mathsf{Until}\ P_3)$

1) After executing any first operation op:
   - Predicate P_3 is satisfied: definition (3) is fulfilled and the verification stops, or
   - Predicates P_2 and P_3 are not satisfied: definition (3) is not violated yet. The verification process must continue because neither P_2 nor P_3 is true.
   - otherwise, definition (3) is violated. That case is represented by dashed lines and a black state in Figure 2 and denotes the forbidden behavior.
2) If state $(\neg P_2 \wedge \neg P_3)$ is reached after executing a first operation, 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 property is satisfied for the first operation executed: thus, for any operation op

$$\forall(\vec{x}, \vec{y}, \vec{v}).(P_1 \wedge Pre(op) \Rightarrow [S_{op}](\neg P_2 \vee P_3)) \tag{4}$$

where $\vec{x}$ denote the values of the machine variables $(x_1, \ldots, x_n)$, $\vec{y}$ 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 $\vec{x}$, and $\vec{v}$ denote the parameters of operation *op*.

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

$$\forall(\vec{x}, \vec{y}, \vec{v}).(\neg P_2 \wedge Pre(op) \Rightarrow [S_{op}](\neg P_2 \vee P_3)) \tag{5}$$

Let us stress that, contrary to (4) that should always be satisfied, (5) should be satisfied only on states belonging to $From\_To(P_1, P_3)$. However predicate $\neg P_2$ may be larger than set $From\_To(P_1, P_3)$, 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 (5) for operation **Take**:

$$\forall(loan, reservation, me_1, me_2, bo_1, me, bo).$$
$$\left(\begin{array}{c} bo_1 \mapsto me_2 \notin loan \wedge Pre(\textbf{Take}) \\ \Rightarrow \\ [S_{\textbf{Take}}]\,(bo_1 \mapsto me_2 \notin loan \vee bo_1 \mapsto me_1 \in loan) \end{array} \right)$$

Let us remark that the set of states denoted by predicate $(bo_1 \mapsto me_2 \notin loan)$ includes states such that book bo_1 is available and member me_2 is at the head of the reservation queue, i.e, before member me_1. It is obvious that such states violate the previous proof obligation since it is possible to execute operation **Take** and lend book bo_1 to member me_2 $(me = me_2, bo = bo_1)$. These counterexamples are found using a model checker like ProB or Alloy. 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 member me_2 cannot be before that of member me_1 in the reservation queue, since new reservations are added at the end of the queue. In addition, me_1 remains in the queue until he borrows the book. So, the specifier, given his knowledge of the specification and the counterexample 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:

$$\left(\begin{array}{c} me_1 \notin ran(reservation(bo_1)) \\ \vee \\ \left(\begin{array}{c} me_2 \in ran(reservation(bo_1)) \\ \wedge \\ Index(bo_1, me_2) < Index(bo_1, me_1) \end{array} \right) \end{array} \right)$$

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 $(\mathsf{Abs}(P_2, \mathsf{After}\ P_1\ \mathsf{Until}\ P_3))$ is satisfied iff there exists a predicate P' such that the following proof obligations hold for each operation *op*:

$$(i)\ \ \forall(\vec{x}, \vec{y}, \vec{v}).(P_1 \wedge Pre(op) \Rightarrow [S_{op}](\neg(P_2 \vee P') \vee P_3))$$
$$(ii)\ \ \forall(\vec{x}, \vec{y}, \vec{v}).(\neg(P_2 \vee P') \wedge Pre(op) \Rightarrow [S_{op}](\neg(P_2 \vee P') \vee P_3)) \qquad \blacksquare$$

The proof of this theorem is provided in appendix.

C. *Characterization of $From\_To(P_1, P_2)$*

In this section, we describe the heuristic we propose to characterize set $From\_To(P_1, P_2)$. It is guided by the search of counterexamples for proof obligation (5):

$$\forall(\vec{x}, \vec{y}, \vec{v}).(\neg P_2 \wedge Pre(op) \Rightarrow [S_{op}](\neg P_2 \vee P_3))$$

When a potential counterexample is found, it is analyzed in order to know whether it is a true counterexample or not. A true counterexample corresponds to a state that is reachable from P_1, that is, it belongs to set $From\_To(P_1, P_3)$. If the counterexample is a true one, then we can conclude that the property is not satisfied. Otherwise, the user should define a predicate P' in order to rule out this false counterexample. As said before, this process is repeated until no false counterexample is founded.

From a practical point of view, to determine the potential counterexample, we represent (5) in the following manner. We define an abstract non deterministic initialization of the machine on page 4 that satisfies invariant Inv and predicate $\neg(P_2 \vee P')$ using a nondeterministic substitution of the form $\vec{x}\!: (P)$, which means that values are assigned to variables $\vec{x}$ such that predicate P is true:

$$\vec{x}\!: (Inv \wedge \neg(P_2 \vee P'))$$

Then, we ask the model checker about the truth value of the LTL property: $\mathsf{X}(\neg(P_2 \vee P'))$ that checks whether the next state violates $\neg(P_2 \vee P')$. If so, a counterexample is given $(\vec{x} = \vec{v})$. In order to know if it is a true counterexample or not, we use a similar approach by changing the initialization of variables $\vec{x}$ into: $(\vec{x}\!: (Inv \wedge P_1))$, and asking again the model checker about the truth value of the CTL [6] property: $\mathsf{EF}(\vec{x} = \vec{v})$ that verifies whether there is a path that permits to reach such a state. At that point, two cases are possible:

1) if there is a state σ, belonging to $From\_To(P_1, P_3)$, that satisfies $\mathsf{EF}(\vec{x} = \vec{v})$, then we can conclude that property $\mathsf{Abs}(P_2,\ \mathsf{After}P_1\ \mathsf{Until}\ P_3)$ is not satisfied.
2) otherwise, the counterexample is a false one. So, we have to define predicate P'' that rules it out. The process is then reiterated with $(\neg(P_2 \vee (P' \vee P'')))$.

It is important to note that the LTL and the CTL formulas we submit for model checking are not very complex because they require only few variables to be checked and a small number of instances is usually sufficient to highlight useful counterexamples. Thus, the model checker will not usually suffer from the state explosion problem and will terminate in very reasonable time by giving the verdict. Moreover, the designer who is verifying the property should have in mind the set of states that can be reached from P_1. So, the process we have proposed is more a heuristic for the designer in order to refine and reinforce his knowledge of the system. If a counterexample is provided by the model checker, the designer should analyse it in order to define a more general predicate that rules it out. This is typically not a difficult task for a designer and it is part of the process of analyzing the behavior of the system.

V. Verifying the Assertions

In this section, we illustrate how to use the proposed assertions in B and Alloy and illustrate it on our case study.

A. *Proving the Assertions in B*

Applying the proof rules (i) and (ii), provided in Theorem 1 gives the following proof obligations (POs):

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

where $\vec{v}$ includes the free variables of the absence property $(\{me_1, me_2, bo_1\})$ and the formal input parameters of operation **op**. Predicates P_1, P_2, P' and P_3 are as follows:

$$\begin{array}{lcl}
P_1 & = & \begin{pmatrix} me_1 \in ran(reservation(bo_1)) \\ \wedge \\ me_2 \notin ran(reservation(bo_1)) \\ \wedge \\ bo_1 \mapsto me_2 \notin loan \end{pmatrix} \\
P_2 & = & (bo_1 \mapsto me_2 \in loan) \\
P' & = & \left(\begin{array}{c} me_1 \notin ran(reservation(bo_1)) \\ \vee \\ \begin{pmatrix} me_2 \in ran(reservation(bo_1)) \\ \wedge \\ Index(bo_1, me_2) < Index(bo_1, me_1) \end{pmatrix} \end{array} \right) \\
P_3 & = & (bo_1 \mapsto me_1 \in loan)
\end{array}$$

To be discharged using the prover of Atelier B, proof obligations **PO1** and **PO2** are added as assertions (clause **ASSERTIONS** of the B notation) to machine *Library* of page 4. A formula of this clause must be proved using the invariant of the machine. Table I gives the statistics we obtained on operations **Loan**, **Take**, **Return** and **Reserve**. 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:

$a \in \mathbf{iseq}(b)\ \wedge$

Proof obligation	Automatic Proofs	Interactive Proofs
PO1	0	5
PO2	0	5

Table I
PROOF RESULTS

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

B. Model Checking the Assertions in Alloy

Proving our assertions require expertise and experience with theorems provers. These resources may not be available in a typical development team. Using a model checker is significantly easier than using a prover. As mentioned earlier, conventional model checking of LTL formula over data-intensive systems quickly suffers from combinatorial explosion. For the case study at hand, verification is limited to 3 or 5 books and members, depending on the model checker used. However, when using assertions instead of LTL formula, Alloy becomes very efficient and can solve significantly larger models in a few seconds, thus providing an increased confidence in the correctness of the system. For the library system, this is particularly important: 3 books and 3 members means that we can have one member borrowing the book and only two members reserving it, which is not much to make sure that the reservation queue is properly handled by all operations.

We will briefly illustrate the verification of the case study in Alloy. The system state is represented by signatures as follows:

```
sig Book{}
sig Member{}
sig Lib
{
  members: set Member,
  books:   set Book ,
  loans:   books ->  members,
  reservations: (seq members)->books
}
```

Signature `Lib` represents the states of the system, which contains four variables: `members`, `books`, `loans`, and `reservations`, which is essentially the same as in the B specification. Operators `->` and `seq` respectively denote Cartesian product and sequences.

Operations that define transitions on the system state are represented by predicates. For instance, operation **Take** is defined as follows.

```
pred Take[m:Member,b:Book,L,L':Lib]
{
  -----Preconditon-------
// b and m are in the Library
  (b in Lib.books) and (m in L.members)
// the loan limit is satisfied
  (#((L.loans).m)<Constants.maxNbLoans)
// m is the first reservation
  (L.reservations.b) = (0 -> m)
// the book is not borrowed
  no (b.(L.loan))

  -----PostCondition-----
// add the new loan to loans
  L'.loan = L.loan  + (b->m)
// delete m from the list of
// reservations of b
  L'.reservations.b =
    delete[L'.reservations.b,0]

  -----Nochanges-------
  all b' : Book - b |
    L'.reservations.b' =
    L.reservations.b'
  L'.books = L.books
  L'.members = L.members
}
```

Operator "`.`" denotes relational composition, and becomes quite handy to extract the components of a relation, in a syntax similar to object-oriented langages. Each object in an Alloy specification denotes a relation (sets are considered as unary relations; constants and elements of sets are singleton sets).

The correspondance with the B specification is almost one to one, except for the following differences. The predicate must include in its parameters the before-state (denoted here by `L`) and the after-state (denoted by `L'`). The predicate describes the relationship between the before-state and the after-state. Alloy being a pure first-order logic language, operations must describe what changes and what does not change, whereas substitutions in B allow one to describe only the variables that change (*i.e.*, Alloy suffers from the well-known frame problem). This is usually not an issue, although in some case studies with a large number of state variables, it can become quite cumbersome [4].

Assertions of Theorem 1 are checked as follows in Alloy.

```
check Th1PO1
{
all b,bo:Book,m,me1,me2:Member,L,L':Lib |
    P1[bo,me1,me2,L] and Transition[b,m,L,L']
  =>
    not P2p[bo,me1,me2,L'] or P3[bo,me1,L']
} for  2 Lib, 5  Member, 5  Book,
       4 int, 5  seq

check Th1PO2
{
all b,bo:Book,m,me1,me2:Member,L,L':Lib |
    not P2p[bo,me1,me2,L] and
      Transition[b,m,L,L']
  =>
    not P2p[bo,me1,me2,L'] or P3[bo,me1,L']
} for  2 Lib, 5  Member, 5  Book,
       4 int, 5  seq
```

The command `check` verifies that a formula is valid for all models of the signature previously defined. The clause `for` describes the bounds for the number of signature elements considered. For these two assertions, we check pairs of library states (`2 Lib`), since the assertions are on before and after states, each library state containing up to 5 books and 5 members. The bounds for `int` and `seq` respectively indicates the number of bits used to represent integers and the maximum length of a sequence; with n bits, Alloy checks integers of the range $-(2^{n-1})$ to $2^{n-1}-1$. The bound for the length of sequences must be equal to the number of members, since in the worst case all members can reserve the same book. Predicate `P1[bo,me1,me2,L]` represents predicate P_1 of the absence property, and is defined as follows:

```
pred P1[bo:Book,me1,me2:Member,L:Lib]
{
reservedBy[bo,me1,L]
not reservedBy[bo,me2,L]
not borrowedBy[bo,me2,L]
}
```

Predicates $P_2 \vee P'$ and P_3 are represented in a similar manner. Predicate `Transition[b,m,L,L']` is a disjunction of the operation predicates.

```
pred Transition[b:Book,m:Member,L,L':Lib]
{
     Lend[m,b,L,L']
  or Reserve[m,b,L,L']
  or Take[m,b,L,L']
  or Return[m,b,L,L']
  ...
}
```

Alloy takes less than 0.3 seconds to check each assertion on the complete library specification used in [7]. Figure 3 illustrates the growth in the verification time of PO1 for Theorem 1 for an equal number of books and members. For 36 books and 36 members, Alloy takes 4.4 mins and 665MB of memory using Alloy 4.1 with SAT4J on a 2.10GHz processor.

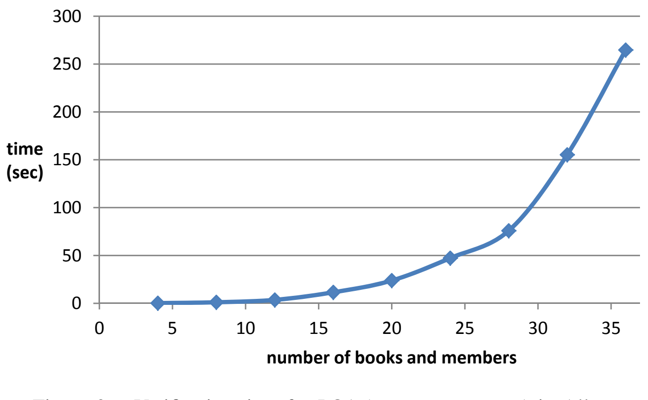

Figure 3. Verification time for PO1 (`check Th1PO1`) in Alloy

VI. Two Variants of the Assertions

In this section, we propose two variants of Theorem 1, which are also sound and complete, to provide other alternatives to prove the absence property. Depending on the specification at hand, one may be easier to use for proofs than the others. For model checking, they all roughly require the same verification time, since the formula are of the same size.

Theorem 2: Let P_1, P_2 and P_3 be three predicates. Property $(\mathsf{Abs}(P_2, \mathsf{After}\ P_1\ \mathsf{Until}\ P_3))$ is satisfied iff there exists a predicate ϕ such that:

(i) $\forall(\vec{x}, \vec{y}).(P_1 \Rightarrow \phi)$

(ii) for each operation op whose parameters are $\vec{v}$: $\forall(\vec{x}, \vec{y}, \vec{v}).(\phi \wedge Pre(op) \Rightarrow [S_{op}]((\phi \wedge \neg P_2) \vee P_3))$ ■

To be able to use this theorem, the user has to propose predicate ϕ according to his/her understanding of the system and the property to prove. To this effect, he/she may start with P_1 as a candidate for ϕ and weaken it to include states of $From\_To(P_1, P_3)$. For instance, P_1 states that me_2 is not in the reservation queue. Of course, executing operation **Reserve** can add me_2 to the reservation queue. These counterexamples can also be found using a model checker and a process similar to the one presented in the previous section. For the running example, predicate ϕ is equal to:

$$\phi = \left(\begin{array}{c} me_1 \in ran(reservation(bo_1)) \\ \wedge \\ bo_1 \mapsto me_2 \notin loan \\ \wedge \\ \left(\begin{array}{c} me_2 \in ran(reservation(bo_1)) \\ \Rightarrow \\ Index(bo_1, me_2) > Index(bo_1, me_1)) \end{array} \right) \end{array} \right)$$

The first condition of Theorem 2 generates less proof obligations than the first condition of Theorem 1, because there is no need to prove it for each operation. However, it may need additional invariants to be proved, and these invariants need to be proved for each operation. Thus, depending on the specification, one theorem may be easier to use than the other. For the case study at hand, condition (i) of Theorem 2 generates no proof obligation because the implication is trivial. The second condition gives rise to 5 POs very similar to those produced for condition (ii) of Theorem 1 and which are discharged by executing the same proof rules and strategies.

The next theorem is the second variant of Theorem 1.

Theorem 3: Let P_1, P_2 and P_3 be three predicates. Property $(\mathsf{Abs}(P_2, \mathsf{After}\ P_1\ \mathsf{Until}\ P_3))$ is satisfied iff there exists a predicate ϕ such that:

(i) $\forall(\vec{x}, \vec{y}).(P_1 \Rightarrow \phi)$

(ii) $\forall(\vec{x}, \vec{y}).(\phi \Rightarrow \neg P_2)$

(iii) for each operation op whose parameters are $\vec{v}$: $\forall(\vec{x}, \vec{y}, \vec{v}).(\phi \wedge Pre(op) \Rightarrow [S_{op}](\phi \vee P_3))$ ■

In this version, we have an additional condition compared to Theorem 2, but it simplifies the third condition. The difference between the two is that ϕ is explicitly proved to be stronger than $\neg P_2$ by condition (ii). For the case study at hand, the candidate for ϕ is the same as the one used for Theorem 2. The generated proof obligations are also the same, since P_2 is a conjunct of ϕ.

VII. Checking Soundness and Completeness of Assertions using Alloy

We have developed three variants of assertions to check the absence property. One may be easier to use than the others for theorem proving, depending on the specification at hand. This raises the issue of rapidly developing sound and complete assertions for other property patterns [5]. Alloy turns out to be very useful to rapidly check the soundness and completeness of assertions for small generic transition models. This helps a designer to quickly check soundness and completeness theorems on small models, and debug them, before formally proving them for arbitrary models. In this section, we briefly illustrate the Alloy specification that can be used for checking soundness and completeness for the absence pattern.

A generic transition system on a set of states `S` is defined by the following signature.

```
sig S {t : set S}
```

This defines `t` as a relation on set `S`. To represent predicates P_1, P_2 and P_3, we define them as subsets of `S`.

```
sig P1,P2,P3 in S {}
```

We define the states reachable from P_1 to P_3 as follows.

```
sig FromP1toP3 in S {}
fact { FromP1toP3 = P1.^(t:>cp[P3]) }
```

A `fact` imposes constraints on the values of symbols in signatures. Function `cp` returns the complement of a set, thus simulating the negation of a state predicate. Operator `^` computes the transitive closure of relation `t` post-restricted (operator `:>`) to $\neg P_3$. Weakest-precondition is represented by the following function.

```
fun wp[r : S->S, Q : set S] : set S
{
  { s : S | s in r.S and s.r in Q}
}
```

The invalidity of the absence property on transition relation `t` is represented by the following predicate.

```
pred Invalid[]
{ some s : S | s in P2 & FromP1toP3 }
```

Predicate `Invalid` holds when some state `s` satisfies `P2` and is in `FromP1toP3` (operator `&` denotes intersection).

The conditions of a soundness and completeness theorem like Theorem 1 are defined as follows.

```
pred Theorem1Conditions[P' : set S]
{
  P1 & t.S in wp[t,cp[P2 + P']+P3]

  cp[P2 + P'] & t.S in wp[t,cp[P2 + P']+P3]
}
```

Finally, soundness and completeness is checked as follows.

```
check Theorem1SoundAndComplete{
  not Invalid[]
<=>
  Theorem1Conditions[cp[FromP1toP3]]
} for 5
```

We have to propose the value of P'. We know that it is the complement of the set of states reachable from P_1, and thus we check the equivalence between the validity of the absence property and the theorem conditions for this value of P'. For sets of 5 states, Alloy takes 0.25 sec to make this verification. We can check Theorem 2 and Theorem 3 in a similar manner, with $\phi =$ `FromP1toP3+P1`. If the theorem is not sound and complete, Alloy provides counterexamples which helps the designer fine tune his/her conditions. Unfortunately, due to solemnization limitations, Alloy cannot automatically compute a value of P' or ϕ, because this requires a second order quantification on sets, which Alloy cannot handle.

VIII. Conclusion

In this paper, we have defined three equivalent sets of necessary and sufficient conditions for proving absence properties of the form $\mathsf{Abs}(P_2,\ \mathsf{After}\ P_1\ \mathsf{Until}\ P_3)$. 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 verifying that the execution of any operation on these states makes the system move to a state verifying P_3 or $\neg P_2$. Guidelines and a practical heuristic are provided in this paper in order to help the user define the set of states reachable starting from any state verifying P_1.

Our approach enables the proof of the absence property using conventional first-order theorem provers in popular state-based methods like ASM, B, VDM, and Z. It also significantly extends the size of models than can be model checked, when compared with traditional LTL model checkers, by using a SAT-based first-order logic model checker like Alloy.

First-order assertion-based approaches for proving other temporal properties have been proposed: see [8], [15] for the *leadsto* modality ($\Box(P \Rightarrow \Diamond Q)$), see [11] for several forms of CTL* formula (*i.e.*, CTL and LTL formula, but not the absence pattern).

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 member will get a book 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] 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

[6] 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.

[7] 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.

[8] T. S. Hoang, J.-R. Abrial, *Reasoning about Liveness Properties in Event-B*. ICFEM 2011, Springer, pp. 456–471.

[9] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, Revised Edition, MIT Press, 2012.

[10] 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.

[11] Kesten, Y., Amir Pnueli, "A Compositional Approach to CTL* Verification", *Theoretical Computer Science*, 331(2-3), pp 397–428, February 2005.

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

[13] M. Leuschel, J. Falampin, F. Fritz and . Plagge, *Automated Property Verification for Large Scale B Models with ProB*, Formal Aspects of Computing, 23(6), 2011 pp. 683–709.

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

[15] J. Misra, *A Discipline of Multiprogramming*, Springer-Verlag, 2001.

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

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

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

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

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

[21] B. Schlich, S. Kowalewski, *Model checking C source code for embedded systems* International Journal on Software Tools for Technology Transfer (STTT), 11(3), Springer, pp. 187–202,

[22] 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

Appendix

Correctness proof

The goal of this section is to demonstrate that Theorem 1 is correct to prove the property pattern $\mathsf{Abs}(P_2, \mathsf{After}\ P_1\ \mathsf{Until}\ P_3))$. In other words, we have to establish that proofs obligations (i) and (ii) are complete and sound.

Are the proof obligations sound?: To establish that the proof obligations are sufficient, let us assume:

1) proof obligations (i) and (ii) are established for predicate $(P_2 \vee P')$ where P' denotes the predicate added by the heuristic previously described.
2) for a given path $(\sigma_0, \ldots, \sigma_{(i-1)}, \sigma_i, \ldots)$ of $Paths(P_1)$ and a rank $i_{(i \geq 1)}$, we have:

$$path[i] \models P_2 \tag{H$_1$}$$

and let us prove that:

$$\exists\, j.(j > 0 \wedge j \leq i \wedge path[j] \models P_3) \tag{G$_1$}$$

Let us establish goal (G$_1$) according to the value of i.

1) <u>case $i = 1$</u>: hypothesis (H$_1$) becomes:

$$path[1] \models P_2 \tag{H$_2$}$$

Let us assume that to move to state $path[1]$, the system executes a given operation op', that is:

$$path[0] \models Pre(op') \wedge path[1] \in S_{op'}(path[0]) \tag{H$_3$}$$

According to proof obligation (i), we have the following for operation op' :

$$\forall(\vec{y}, \vec{x}, \vec{param}).(P_1 \wedge Pre(op') \Rightarrow [S_{op'}](\neg(P_2 \vee P') \vee P_3))$$

Given that, for any predicates P_1, P_2 and any substitution S, expression $(P_1 \Rightarrow [S]P_2)$ is equivalent to:

$$\forall\, \sigma.(\sigma \models P_1 \Rightarrow \sigma \in dom(S) \wedge \forall\, \sigma'.(\sigma' \in S(\sigma) \Rightarrow \sigma' \models P_2))$$

We obtain the following:

$$\forall\, \sigma. \left(\begin{array}{c} \sigma \models (P_1 \wedge Pre(op')) \\ \Rightarrow \\ \forall\, \sigma'. \left(\begin{array}{c} \sigma' \in S_{op'}(\sigma) \\ \Rightarrow \\ \sigma' \models (\neg(P_2 \vee P') \vee P_3) \end{array} \right) \end{array} \right) \tag{H$_4$}$$

By definition of $Paths(P_1)$, we know that:

$$path[0] \models P_1 \tag{H$_5$}$$

Let us instantiate (H$_4$) on states $path[0]$ and $path[1]$, and using hypotheses (H$_3$) and (H$_5$), we obtain:

$$path[1] \models (\neg(P_2 \vee P') \vee P_3) \tag{H$_6$}$$

(H$_2$)+(H$_6$) give:

$$path[1] \models P_3 \tag{H$_7$}$$

Goal (G$_1$) is thus discharged for $(j = 1)$.

2) case $i \geq 2$: let l $(l \leq i)$ be the first instant such that $path[l] \models P_2$, that is:

$$path[l] \models P_2 \wedge \forall\, k.(k \in 1..(l-1) \Rightarrow path[k] \models \neg P_2) \tag{H$_8$}$$

Let us assume that to move to state $path[l]$, the system executes a given operation op', that is:

$$path[l-1] \models Pre(op') \wedge path[l] \in S_{op'}(path[l-1]) \tag{H$_9$}$$

According to proof obligation (ii), we have the following for operation op':

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

which can be rewritten into :

$$\forall\, \sigma. \left(\begin{array}{c} \sigma \models (Pre(op') \wedge \neg(P_2 \vee P')) \\ \Rightarrow \\ \forall\, \sigma'. \left(\begin{array}{c} \sigma' \in S_{op'}(\sigma) \\ \Rightarrow \\ \sigma' \models (\neg(P_2 \vee P') \vee P_3) \end{array} \right) \end{array} \right) \tag{H$_{10}$}$$

It is obvious that if there is a rank m such that:

$$m \in 1..l \wedge path[m] \models P_3$$

then, (G$_1$) will be discharged for $(j = m)$. So, let us assume that

$$\forall\, k.(k \in 1..l \Rightarrow path[k] \models \neg P_3) \tag{H$_{11}$}$$

We can easily prove by induction and using proof obligation (ii) of Theorem 1 that :

$$\forall\, k.(k \in 1..l \Rightarrow path[k] \models (\neg(P_2 \vee P') \vee P_3) \tag{H$_{12}$}$$

(H$_{11}$)+ (H$_{12}$) give:

$$\forall\, k.(k \in 1..l \Rightarrow path[k] \models \neg P') \tag{H$_{13}$}$$

By instantiating (H$_{10}$) with $path[l-1]$ and $path[l]$ respectively, (H$_{13}$) by $path[l-1]$ and (H$_8$) by $path[l-1]$ and using hypothesis (H$_9$), we obtain:

$$path[l] \models (\neg(P_2 \vee P') \vee P_3) \tag{H$_{14}$}$$

(H$_8$)+(H$_{14}$) give:

$$path[l] \models P_3 \tag{H$_{15}$}$$

Goal (G$_1$) is thus discharged for $(j = l)$.

Are the proof obligations complete?: Proving that the proof obligations (i) and (ii) are complete comes down to demonstrating that if these proof obligations are not verified for any predicate P', then we can conclude that property $\mathsf{Abs}(P_2, \mathsf{After}\ P_1\ \mathsf{Until}\ P_3))$ is false. To do that let us suppose that:

$$\forall P'.(\neg PO_i \vee \neg PO_{ii}) \tag{H$_{16}$}$$

where PO_i and PO_{ii} denote the proof obligations i and ii of Theorem 1, and let us prove that

$$\exists(path, i).\left(\begin{array}{c} path \in Paths(P_1) \\ \wedge \\ path[i] \models P_2 \\ \wedge \\ \forall j.\left(\begin{array}{c} j > 0 \wedge j \leq i \\ \Rightarrow \\ path[j] \models \neg P_3 \end{array}\right) \end{array}\right) \tag{G$_2$}$$

Let predicate P_{NRS} be the set of the states that verify the following two conditions:

1) cannot be reached from any state verifying P_1,
2) verify predicate P_2 or there exists a path from this state to a state that verifies predicate P_2.

Instantiating (H$_{16}$) with P_{NRS} gives for a given operation *op*:

$$\exists(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{param}).(P_1 \wedge Pre(op) \wedge \neg[S_{op}](\neg(P_2 \vee P_{NRS}) \vee P_3)) \tag{H$_{17}$}$$

or

$$\exists(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{param}).(\neg(P_2 \vee P_{NRS}) \wedge Pre(op) \wedge \neg[S_{op}](\neg(P_2 \vee P_{NRS}) \vee P_3)) \tag{H$_{18}$}$$

So, we have to deal with two cases:

- <u>Hypothesis (H$_{17}$) is true</u>: (H$_{17}$) can be rewritten into:

$$\exists(\beta, \beta').(\beta \models (P_1 \wedge Pre(op)) \wedge \beta' \in S_{op}(\beta) \wedge \beta' \models ((P_2 \vee P_{NRS}) \wedge \neg P_3)) \tag{H$_{19}$}$$

  Let $(\beta_0, {\beta_0}')$ be values of (β, β') that verify (H$_{19}$):

$$(\beta_0 \models (P_1 \wedge Pre(op)) \wedge {\beta_0}' \in S_{op}(\beta_0) \wedge {\beta_0}' \models ((P_2 \vee P_{NRS}) \wedge \neg P_3))$$

  Since ${\beta_0}'$ is reachable from P_1 and P_{NRS} denotes the set of the states that are not reachable from P_1, ${\beta_0}'$ does not verify P_{NRS}. So, ${\beta_0}'$ verifies P_2 and goal (G$_2$) is verified for $((path, i) = ((\beta_0, {\beta_0}'), 1))$.

- <u>Hypothesis (H$_{18}$) is true</u>: (H$_{18}$) can be rewritten into:

$$\exists(\beta, \beta').\left(\begin{array}{c} \beta \models (Pre(op) \wedge \neg(P_2 \vee P_{NRS})) \\ \wedge \\ \beta' \in S_{op}(\beta) \wedge \beta' \models ((P_2 \vee P_{NRS}) \wedge \neg P_3)) \end{array}\right)$$

  Let $(\beta_0, {\beta_0}')$ be values of (β, β') that verify this last predicate:

$$\left(\begin{array}{c} \beta_0 \models (Pre(op) \wedge \neg(P_2 \vee P_{NRS})) \\ \wedge \\ {\beta_0}' \in S_{op}(\beta_0) \wedge {\beta_0}' \models ((P_2 \vee P_{NRS}) \wedge \neg P_3) \end{array}\right) \tag{H$_{20}$}$$

  If state β_0 is reachable from a state verifying P_1, that is:

$$\exists(\alpha_0, \ldots, \alpha_n). \left(\begin{array}{c} \alpha_0 \models P_1 \\ \wedge \\ \bigwedge_{i=1,n} \alpha_i \in From\_To(P_1, P_3) \\ \wedge \\ \alpha_n = \beta_0 \end{array} \right)$$

In that case, ${\beta_0}'$ is also reachable from P_1. Since P_{NRS} denotes the set of the states that are not reachable from P_1, ${\beta_0}'$ does not verify P_{NRS}. So, ${\beta_0}'$ verifies P_2 and goal (G$_2$) holds for $(path, i) = ((\alpha_0, \ldots, \alpha_n, {\beta_0}'), n + 1)$.

Now, let us assume that β_0 is not reachable from P_1. If ${\beta_0}'$ verifies P_2 , β_0 would verify P_{NRS} because ${\beta_0}'$ is reachable from β_0. This would contradict (H$_{20}$). Otherwise, ${\beta_0}'$ should verify P_{NRS} , then β_0 would verify P_{NRS} because ${\beta_0}'$ is reachable from β_0. This would also contradict (H$_{20}$).

■