Comparison of Model Checking Tools for Information Systems

Marc Frappier Benoît Fraikin Romain Chossart Raphaël Chane-Yack-Fa Mohammed Ouenzar

GRIL, Université de Sherbrooke, Québec, Canada

June 16, 2010

Abstract

This paper compares six model checkers (Alloy, cadp, fdr2, NuSMV, ProB, Spin) for the validation of information system specifications. The same case study (a library system) is specified using each model checker. Fifteen properties of various types are checked using temporal logics (CTL and LTL), first-order logic and failure-divergence (fdr2). Three characteristics are evaluated: ease of specifying information system i) behavior, ii) properties, and iii) the number of IS entity instances that can be checked. The paper then identifies the most suitable features required to validate information systems using a model checker.

Contents

List of Figures

Listings

1 Introduction

Information systems (IS) now play a prominent role in our society to support business processes and share organisational data. Yet, even if they are one of the early application domains of computing, their development relies mostly upon a manual and informal process. The problem addressed in this paper is the formal *validation* of IS specifications using model checking. Model checking is an interesting technique for IS specification validation for several reasons: it provides broader coverage than simulation or testing, it requires less human interaction than theorem proving, and it has the ability to easily deal with both safety and liveness properties.

The validation of IS specification is of particular interest in model-driven engineering (MDE) and generative programming, which aim at synthesizing an implementation of a system from models (i.e., specifications). Hence, if the synthesis algorithms are correct, one only needs to validate the models to produce correct systems. IS MDE specification languages usually do not have any dedicated model checker. Since developing a model checker is a long process and since several model checkers already exist, it is simpler to choose an existing tool that is maintained by a team specialized in the model checking field. In this paper, we compare six model checkers: Spin [11], NuSMV [4], fdr2 [18], cadp [10], Alloy [12] and ProB [13], which are representative of the main classes of model checkers: explicit state, symbolic, bounded and constraint satisfaction. The comparison is based on a single case study which is representative of IS structure and properties. Our comparison aims at answering the following questions.

1. Is the modeling language of the model checker adapted for the specification of IS models? This is especially important in the context of MDE IS, since it must be straightforward to automatically translate an IS MDE specification into the language of the model checker.
2. Is the property specification language adapted to specify IS properties?
3. Is the model checker capable of checking a sufficient number of instances of IS entities?

Our case study focuses on the control part of an IS, which determines the sequences of actions that the IS must accept. Validation of input-output behavior (data queries) and user interactions with graphical user interface are omitted.

This paper is structured as follows. Section 2 presents a synthesis of related work on model checking of IS. Section 3 presents a description of the case study, a library IS. Section 4 provides an overview of the model checkers, comparing relevant points. The modeling and verification process of the IS for each tool is provided in Sect. 5. Then, the analysis of processes and the model checking results for the case study are presented in Sect. 6. Finally, we conclude in Sect. 5.

2 Related Work

There is an extensive literature on model checking. This section focuses on comparative studies of model checkers related to IS. Model checking has been extensively applied to business process modelling. Yeung [20] proposes a framework to analyse *suspendible business* transactions modelling with statecharts and csp [18]. It is applied to a library specification similar to the one studied in this paper. However, the paper does not actually experiment with model checkers to check business process. In [2], a travel agency business process has been modelled with Spin and Promela, and CIA and csp (LP). Safety properties and deadlocks have been successfully verified with both model checkers; reachability properties have not been tested. The authors propose an extension of csp with notion of “compensation” (a behavior to compensate a process failure). In [16], business processes are converted from BPEL to automata, but also to Petri nets and csp and lotos [5]. It concludes that process algebras are suitable for verification of the reliability of IS, in the particular case of business process. In [8], the authors study the verification of data-driven applications in the particular case of web-based systems using an ASM-like [19] specification language. The study focuses particularly on reachability properties, but any type of property can be used for modelling. The modelling process is complex and demands significant expertise. Both modelling techniques give an insight on what can be done with these subclasses of IS. In [3], four state-based model modeling techniques

with their model checkers (USE, Alloy, ZLive and ProZ) are compared along four criteria: animation, generation of pre and postconditions, execution analysis and expertise. The study mostly checks invariant properties.

Each of these studies provides partial answers to our questions, for a subset of model checkers, using different case studies and properties. This makes it difficult to compare model checkers and identify the one best suited for IS validation.

3 Presentation of the Case Study

This section presents the user requirements of a library system which is used for the formal verification of properties. In order to avoid any confusion, key concepts are first defined. *Lending* a book means that a user borrows a book without reserving it beforehand. *Taking* a book means borrowing a book after having reserved it. *Borrowing* a book denotes either *taking* or *lending* it. In the requirements list, a *member* is a person who has *joined* (and still not *left*) library membership.

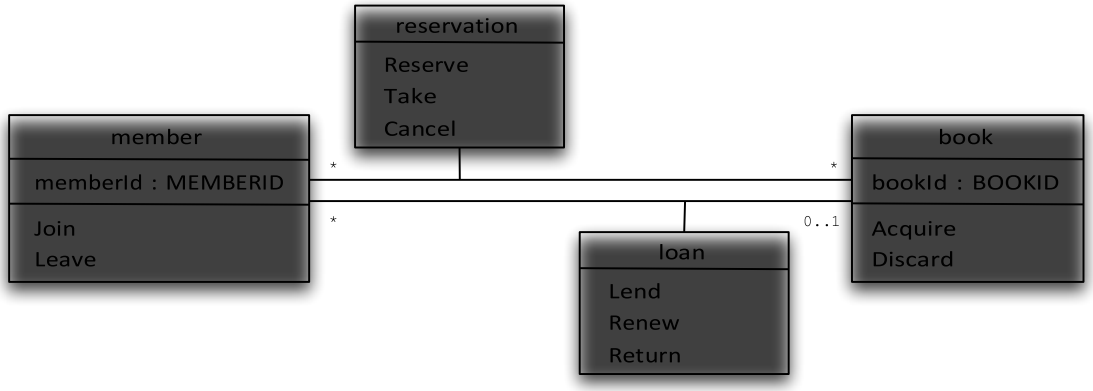

Figure 1: Requirement class diagram of the library system

The requirement class diagram corresponding to the model is given in Fig. 1. Entity attributes are listed in the upper part of each class, while entity actions are listed in the lower part. The library system only contains two entity types: books and members. Members can Join and then Leave library membership whereas books can be Acquired and then Discarded. Members can Lend, Renew several times and Return a book. They can also Reserve a book under certain conditions (e.g. if it cannot be lent at that moment), and then, either Cancel the reservation or Take the book. Hence the library system contains 10 actions.

The following list describes the properties that we verify using the model checkers.

1. A book can always be acquired by the library when it is not currently acquired.
2. A book cannot be acquired by the library if it is already acquired.
3. An acquired book can be discarded only if it is neither borrowed nor reserved.
4. A person must be a member of the library in order to borrow a book.

5. A book can be reserved only if it has been borrowed or already reserved by another member.
6. A book cannot be reserved by the member who is borrowing it.
7. A book cannot be reserved by a member who is reserving it.
8. A book cannot be lent to a member if it is reserved.
9. A member cannot renew a loan if the book is reserved.
10. A member is allowed to take a reserved book only if he owns the oldest reservation.
11. A book can be taken only if it is not borrowed.
12. A member who has reserved a book can cancel the reservation at anytime before he takes it.
13. A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or canceled.
14. Ultimately, there is always a procedure that enables a member to leave the library.
15. A member cannot borrow more than the loan limit defined at the system level for all users.

In the context of IS, one usually distinguishes two types of properties. The first one is called a *liveness* property. It represents the fact that the system is still alive, i.e. not stuck in a deadlock (the system is blocked in a single state) or a livelock (the system loops in a subset of states considered as non-evolving). They can also express the fact that an action implies a reaction from the system; the latter is however rarely used in information systems, where actions are human-driven (one cannot force a user to trigger specific actions). Properties 1, 12 and 14 are liveness properties. In IS, liveness properties usually describe *sufficient conditions* to *enable* an action (immediately or sometime in the future). For instance, Property 1 states that the library *has the right* to acquire the book (under certain conditions). In other terms, it forces the IS to allow the action, but the user is not forced to invoke this action. The other properties are called *safety* properties. They usually describe *necessary conditions* to enable an action, or what a user is *not allowed* to do with the system at a given point. The remaining properties are safety properties. A third type of properties is usually distinguished from these two. These are *fairness* properties, but as IS users cannot be forced to do some action, they seldom occur in IS specifications. Fairness is not considered in this study.

4 An Overview of the Model Checkers

Four large families of model checkers are considered. *Explicit state* model checkers, like cadp, Spin and fdr2, use an explicit representation of the transition system associated to a model specification. This transition system is either computed prior to property verification, as in cadp and fdr2, or on-the-fly while checking a property, as in Spin (also possible in some cases with cadp and fdr2). *Symbolic* model checkers, like NuSMV, represent the transition system as a Boolean formula. *Bounded* model checkers, like NuSMV and Alloy (indirectly), consider traces, of a maximal length k, of the transition system and represent them using a Boolean formula. *Constraint satisfaction* model checkers, like ProB, use logic programming to verify formula. Spin, cadp, NuSMV and ProB support temporal languages (LTL [17], CTL [7] and XTL [14]) for

property specification. Alloy and fdr2 use the same language for both model specification and property specification (first-order logic and csp, respectively).

4.1 Spin

Spin was one the first model checker developed, starting in 1980. It introduced the classical approach for onthe-fly LTL model checking. Specifications are written in Promela and properties in LTL. An LTL property is compiled in a Promela never claim, i.e. a Büchi automaton. Spin generates the C source code of an on-the-fly verifier. Once compiled, this verifier checks if the property is satisfied by the model. Working on-the-fly means that Spin avoids the construction of a global state transition graph. However, it implies that transitions are (re-)computed for each property to verify. Hence, if there are *n* properties to verify, a transition is potentially computed *n* times, depending on optimizations.

Promela, the model specification language of Spin, is inspired from C. Hence, it is an imperative language, with constructs to handle concurrent processes. State variables can be global and accessed by any process. Promela offers basic types like char, bit, int and arrays of these types. Processes can communicate by writing and reading over a *channel*, either synchronously using a channel of length 0, or asynchronously, using a channel of length greater than 0. Operator atomic allows a compound statement to be considered as a single atomic transition, except when this compound contains a blocking statement, such as a guard or a blocking write or read over a channel, in which case the execution of the atomic construct can be interrupted and control transferred to another process. Statements can be labeled and these labels can be used in LTL formulae.

Spin uses propositional LTL, with its traditional operators *always*, *eventually* and *until*. The latter is sometimes referred as the "strong until" operator, as opposed to the "weak until" operator. The *next* operator is not allowed to ensure that partial order reduction can be used during the model checking. An LTL formula can refer to labels and state variable values of a Promela specification. Spin only considers states; there is no notion of event on a transition. An LTL formula holds for a Promela specification if and only if it holds for every possible run of the Promela model. A run is an execution trace consisting of the sequence of states visited during execution. It can be infinite.

4.2 NuSMV

NuSMV is a model checker based on the SMV (Symbolic Model Verifier) software, which was the first implementation of the methodology called *Symbolic Model Checking* described in [15]. This class of model checkers verifies temporal logic properties in finite state systems with "implicit" techniques. NuSMV uses a symbolic representation of the specification in order to check a model against a property. Originally, SMV was a tool for checking (Computational Tree Logic) CTL properties on a symbolic model. But NuSMV is also able to deal with LTL (+Past) formulae and SAT-based *Bounded Model Checking*. The model checker allows to write properties specification both in CTL or LTL and to choose between BDD-based symbolic model checking and bounded model checking.

NuSMV uses the SMV decription language to specify finite state machines. A specification consists of module declarations and each module may include variable declarations and constraints. System transitions are modelled by assignment contraints or transition constraints, which define next values for declared variables in a module. An assignment gives explicitly a value for a variable in the next step, while a transition constraint, given by a boolean formula, restricts the set of potential next values. Each module can be instantiated by another one, for example by the main module, as a local variable. In fact, each

instance of a module is by default processed synchronously with the others during an execution. But NuSMV can also model interleaving concurrency by using the "process" keyword in module instantiation. To get different instances of module, instantiations can be parameterized. However, the description language is quite low-level. All assignments, parameters or array indexes have to be constant. Thus, specifications may be longer than in Promela, because each case has to be explicitly written. As NuSMV modules can declare state variables and input variables, an SMV specification can be both state or event oriented. Input variables are used to label incoming transitions and their values can only be determined by specifying transition constraints.

CTL properties can only be expressed by using state variables, but NuSMV allows to use input variables and state variables in LTL specifications. Moreover, NuSMV can also check invariant properties, which can be written in a temporal manner as *Always p* where *p* is a boolean formula. Invariant specifications are checked by a specialized algorithm during reachability analysis, that gives a result faster than CTL or LTL algorithms.

4.3 fdr2

fdr2 is an explicit state model checker for Communicating Concurrent Processes (csp), the well known process algebra. fdr2 can check refinement, deadlocks, livelocks and determinacy of process expressions. It gradually builds the state-transition graph, compressing it using state-space reduction techniques, while checking properties, which also makes it an *implicit* state model checker.

Models are described using a variant of csp, called CSP_M. It supports classical process algebra operators like prefix, choice, parallel composition with synchronisation, sequential composition and guards. Quantified versions of choice, parallel composition and sequential are supported. fdr2 supports basic data types like integer, boolean, tuples, sets and sequences. Lambda terms can be used to define functions on these types. Expressions are dynamically typed (except for actions, called channels in csp, which are declared and typed). csp does not support state variables; however, they can be simulated to some extent by using a recursive process with parameters.

Properties are expressed as csp processes. They are checked using process refinement. fdr2 supports three refinement relations: v_T (trace refinement), v_F (stable-failure refinement) and v_{FD} (stable-failuredivergence refinement). We say that $P\ \mathrm{v}_T\ Q$ iff the traces of *Q* are included in the traces of *P*. A trace of a process *P* is a sequence of visible events that *P* can execute. We say that $P\ \mathrm{v}_F\ Q$ iff the failures of *Q* are included in the failures of *Q*. A failure (*t*,*S*) of a process *P* denotes the set of events *S* that *P* can refuse after executing trace *t*. Trace refinement is used to check safety properties, while stable failure is used to check liveness (or reachability) properties. Failure-divergence refinement is used to check livelocks (infinite loops on internal actions), which are not relevant for our case study.

4.4 cadp

cadp is a rich and modular toolbox. We have selected lotos-nt to specify models and XTL to specify properties. The XTL model checker takes as input a labelled transition system (LTS), encoded in the BCG (*Binary Coded Graph*) format. lotos-nt is a variant of lotos that supports local states variables. A lotos-nt specification is translated into into a LTS using Caesar. This LTS is minimized into a trace equivalent LTS. Finally, properties written in XTL are checked against this LTS using the XTL model checker.

lotos-nt is inspired from lotos. A lotos-nt specification is divided into two complementary parts: an algebraic specification of the abstract data types and a process expression. lotos-nt offers traditional process algebra operators like sequence, choice, loops, guard and parallel synchronization. It supports state variables, which are local to a process and cannot be referred by another process. Assignment statements can be freely mixed with other process expression constructs.

XTL, the property specification language of cadp, is used to express temporal logic properties. XTL provides low-level operators which can be used to implement various temporal logics like HML, CTL, ACTL, LTAC, as well as the modal mu-calculus. XTL formulae are evaluated on a LTS. XTL allows one to refer to transitions (events) and values of their parameters. No LTL library is currently provided. In this paper, the CTL and HML libraries are used.

Since the LTS does not contain any state variable, the difficult part in writing XTL properties for lotos-nt models is to characterize states. Indeed, the specifier can only use action labels to define particular states. The HML library, with its two handy operators *Dia* and *Box*, is used for this purpose. *Box*(*a*, *p*) holds in a given state if and only if every action matching action pattern *a* leads to a state matching state pattern *p*. On the other hand, *Dia*(*a*, *p*) holds for a given state if and only if there exists at least one action matching action pattern *a* leading to a state matching state pattern *p*. An XTL formula holds for a LTS if and only if it holds for all states of the LTS.

4.5 Alloy

Alloy is a symbolic model checker. Its modeling language is first-order logic with relations as the only type of terms. Basic sets and relations are defined using "signatures", a construct similar to classes in object-oriented programming languages, which supports inheritance. Alloy uses SAT-solvers to verify the satisfiability of axioms defined in a model and to find counterexamples for properties (theorems) which should follow from these axioms.

An Alloy specification consists of a set of signatures, noted (sig), which basically define sets and relations. Constraints, noted fact, are formulae which condition the values of sets and relations. The declaration sig X {r : X -> Y} declares a set X and a ternary relation r which is a subset of the Cartesian product $X \times X \times Y$. Alloy supports usual operations on relations, like union, intersection, difference, join, transitive closure, domain and range restriction. Integer is the only predefined type. Cardinality constraints can be defined on relations (*e.g.*, injections and bijections). Properties are simply written as first-order formulae.

4.6 ProB

ProB is a model checking and an animation tool designed for the B Method [1]. Currently it also supports CSP_M, Z, and Event-B. This study uses the B Method and the B language.

B specifications are organized into abstract machines (similar to classes and modules). Each machine encapsulates state variables, an invariant constraining the state variables, and operations on the state variables. The invariant is a predicate in a simplified version of the ZF-set theory, enriched by many relational operators. In an abstract machine, it is possible to declare abstract sets by giving their name without further details. This allows the actual definition of types to be deferred to implementation. Operations are specified in the Generalized Substitution Language, which is a generalization of Dijkstra's guarded command notation. Hence, operations are defined using substitutions, which are like assignment

statements. A substitution provides the means for identifying which variables are modified by the operation, while avoiding mentioning those that are not. The generalization allows the definition of non-deterministic and preconditioning substitutions. The preconditioning substitution is of the form PRE *P* THEN *S* END, where *P* is a predicate and *S* a substitution. When *P* holds, the substitution is executed; otherwise, the result is undetermined and the substitution may abort.

Properties in ProB can be written in LTL, past LTL or CTL, hence combining the strengths of each language. In addition, ProB allows for the inclusion of first-order formulae in temporal formulae. It also offers two convenient operators for LTL. The first one, denoted by $e(A)$, checks if the action *A* is executable in a given state of a sequence. The second one, denoted by $[A]$ checks if *A* is the next operation in the sequence. Consequently ProB can express properties on either states or events.

5 Specifying the Model and the Properties

This section describes how the IS model and properties are specified with each model checker. For sake of conciseness, specifications are omitted. They are available in [6].

5.1 Spin

Two styles have been considered for the Spin specification. In the first style, there is only one process which loops over a choice between all actions. It was quickly abandoned, because it blows quite rapidly in terms of number of states. In the second style, there is one process for each instance of each entity and each association. The process describes the entity (or the association) life cycle. Therefore, the Promela specification of the case study contains four process definitions, one for each entity (book and member) and one for each association (loan and reservation). Each process definition is instantiated n_i times to model n_i instances of entity *i*, and $n_i * n_j$ times to model an association between entity *i* and *j*.

We use a producer-modifier-consumer pattern as the basis of a life cycle for an entity and an association. It can be represented by the following regular expression $P.M^*.C$ where *P* is the producer (for example Acquire), *M* is a modifier and *C* is a consumer (for example Discard). The concatenation operator "." of regular expressions can be represented by a semi-colon ";", the sequential composition operator of Promela, or an arrow "->" that denotes the same operator. Some events have a precondition which is not represented in the regular expression. For example, a book cannot be discarded if it is still borrowed. Consequently the execution of an event is guarded by a precondition.

When a member takes a book he has reserved, two associations are involved: the loan association and the reservation association. This leads to ensure that both processes execute the take event in an atomic step. It is not obvious and straightforward. To achieve an atomic step, the take event is split into two events: one in the reservation association process (as a consumer) and one in the loan reservation (as a producer). A channel with an empty capacity is used to ensure the handshake. This is a classic strategy in Promela. Nevertheless, the handshake cannot be made within an atomic instruction. This could break the local atomicity in the sender. But it could be used at the end of an atomic and at the beginning of another. In this way, no other process can be interleaved with the handshake of the two processes. The result is a pattern described in [6], in which an event is simultaneously the consumer of an association and the producer of another one.

In Spin, the properties are expressed using LTL. Reachability properties are difficult to express in LTL. Fortunately, since event preconditions are explicitly written via labels in the specification, expressing a

property like “a book can be acquired” is straightforward. Consequently, when a property asserts the possibility of an event execution, it is represented by a propositional formula in the LTL formula that uses a label of the process and, sometimes, the precondition of this event. For example, “the book b0 can be acquired” is expressed as “process book b0 is at the discarded label”.

Property 14 is not expressible in LTL, since it is equivalent to a reset property. The reset property is known to be expressible in CTL only. LTL and CTL are complementary languages. The semantics of LTL formulae is defined on traces of the transition system, while the semantics of CTL formulae is defined on the transition system itself, which allows one to refer to the branching structure of the execution. Some properties can only be expressed in either LTL or CTL. For instance, a *reset* property, which states that it is always possible to go back to some desired state, cannot be expressed in LTL, since this transition to reset does not have to occur in each possible run of the system. Since an LTL formula holds if and only if it holds for every possible run of the system, an LTL property would force this reset transition to occur in every run. Dually, a property of the form “eventually p always holds” cannot be expressed in CTL [9], due to the branching nature of the logic.

Two simple patterns can express almost 70% of the requirements. In LTL and with the state-oriented paradigm, these patterns are expressed as “if action *A* can be executed then the state verifies *P*” or “if *P* holds then action *A* can be executed” where *P* is only true between *B* and *C*. Therefore the two main patterns are can_*A* ⇒ *P* or *P* ⇒ can_*A*. Inexpressible properties cannot simply be considered as negligible. This is an important weakness of Spin that cannot be overcome.

5.2 NuSMV

To model the library system example in an SMV specification, we use a systematic method based on the structure of the class diagram. Each class, that represents an object and has attributes, is encoded into a module containing variables and parameterized by a key to identify entities. Then, for each kind of action defined in the system, a new module is created, parameterized by class modules involved in that action. Action modules check that a given precondition is satisfied. Then, if the precondition holds, they modify variables of entities to apply postconditions using assignment constraints.

Properties of the library system can be expressed by CTL formulae on state variables, or by using LTL formulae with state variables or input variables. Specifications on state variables are close to Promela specifications, except that NuSMV can check CTL and LTL properties. This allows to easily express all requirements. Specifications on input variables are event-oriented. However, only LTL can be used to write event-oriented specifications in NuSMV.

Property 1 is a sufficient condition to enable an event. It is easily expressed as follows: AG (!book1.is_acquired -> EX book1.is_acquired). Property 12 is specified in a similar manner, except that it must be repeated for each position in the array representing the reservation queue. Hence, the text of properties may linearly grow with the number of entity instances, an unfortunate limitation due to the restriction to constants in accessing array positions. Property 14 is also very similar to 1, except that EF is used instead of EX. Properties expressing necessary conditions (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13) can be expressed in two different forms (state, event). For instance, property 2 is expressed using events, saying that a discard must always occur between two acquires:

G acquire1.do -> X((!acquire1.do U discard1.do) | G !acquire1.do)

5.3 fdr2

CSP is quite handy to explicitly represent IS entities life cycles. We use a nominal/controller pattern to specify the library case study. Each entity *E* is represented by a quantified interleave of the form $E = k : T @ E(k)$, where $E(k)$ models the life cycle of an instance *k* of entity *E*. The associations to which an ^{9} entity *E* participates are represented in a similar manner, and called within $E(k)$. The global behavior is obtained by composing the entities in parallel: $Nominal = E_1 \,|[\,]|\, ... \,|[\,]|\, E_n$. Since CSP_M does not directly implement Hoare's parallel operator ||, we use CSP_M's operator $|[X]|$, where *X* denotes the association actions on which entities must synchronize. Some ordering constraints are represented using recursive processes to simulate state variables. The relevant state variables ~*v* of an entity *E* are represented by a recursive process $C_E(k,\sim v)$ which offers a choice [] between actions to control, in the form $[]_i\, G \,\&\, a_i \rightarrow C_E(k, e(\sim v))$, where "&" denotes a guard operator with condition *G* which tests the values of ~*v*. These control processes are composed in parallel with *Nominal*, synchronizing on a_i.

Safety properties are checked by trace refinement and are relatively easy to specify. Suppose that property *p* only involves actions a_i. One writes a process *P* that represents the traces on a_i satisfying *p*, and checks that the system *Q*, restricted to actions a_i, trace refines *P* as follows: $P \sqsubseteq_T Q \setminus (\Sigma \setminus \{a_i\})$, where \ is the hiding operator of CSP_M and Σ denotes all actions of the specification and "–" is set difference.

Reachability properties 1, 12 and 14 are checked using stable-failure refinement, and are a bit more tricky to specify. For instance, property 1 states that a book can always be acquired, if it is not currently in the library. This is typically specified in csp as follows: $P \; CHAOS(\{b_i\}) \sqsubseteq_F Q \setminus \{c_i\}$. Process *P* recursively loops over acquire and discard events: $\text{acquire}(b) \rightarrow \text{discard}^{9}(b) \rightarrow P(b)$. Essentially, *P* states that discard can never be refused after an acquire, and an acquire can never be refused after a discard. Hidden actions $\{c_i\}$ are those that unavoidably can occur between acquire and discard, *ie*, association actions. The interleave with $CHAOS(\{b_i\})$ states that other actions can occur before or after, but we do not really care about their order. Unfortunately, hiding association actions introduces unstable states, which weakens the specification of the property under stable-failure refinement. To make a short explanation, infinite internal action loops are introduced by hiding; hence some errors in the behavior of association actions are not detected by this form of property specification. To overcome this, we have to check each association in isolation, disabling events from the other associations, which is weaker than property 1. These are very subtle issues which are difficult to master. Reachability properties of IS specifications are far from trivial to specify in csp.

5.4 cadp

The lotos-nt specification of the library system is similar in structure to the csp specification already described. Since there is no quantified interleave operator in lotos-nt, one has to hardcode entities and associations interleaves, which means that the number of interleaves to hard code in the specification text grows exponentially with the number of entities, making verification experiments a bit cumbersome.

Safety properties of the case study are defined using two patterns. The first states that an action A should not happen between two actions B and C. For example, a member should not leave the library if he has reserved a book (i.e. between a Reserve and a Take or Cancel). The second pattern expresses the prohibition of an action A outside a sequence delimited by two actions B and C; it is illustrated by the fact that a member should not renew a book if he has not borrowed it yet (i.e. outside a Lend or Take and a Return). In XTL, one can represent these patterns using macros. They are defined using a weak until operator, defined by macro AW_A_B. These two patterns, respectively called no_A_between_B_and_C for

and no_A_outside_B_and_C, are used for properties (2,3,6,7,8,9,11,13) and 4, respectively. Liveness properties are written directly with classic ACTL and HML operators, like properties 1, 12 and 14. No correct formulation has been found for properties 5 and 10. For property 5, one must characterize using events the states where a book is *not borrowed nor reserved*. Property 10 involves a queue, which is as hard to describe using events.

5.5 Alloy

Each IS entity *E* is represented by a signature E, which models the set of possible entity instances. System states are represented by a signature sig S { e1 : E1, ..., en : En, a1 : Ei ->

Ej, ...}, where e_i models the active instances of E_i in a state, and a_i models the instances of association A_i. Each action is represented by a predicate $P[s : S, s^0 : S, p : T]$ relating a before-state s to an after-state s^0 for input parameters p. We have systematically followed a pattern for these predicates, which is a conjunction of a precondition, a postcondition and a "nochange" predicate that determines which attributes are unchanged by the action.

A property of the form "when condition *C* holds, action *a* must be executable" (*e.g.*, Property 1) is written as follows: $\forall s : S, p : T \cdot C \Rightarrow preA[s, p]$, where $preA[s, p]$ is the precondition of action $a(p)$. Similarly, if *C* is the result of executing an action $b(p)$ that should enable an action $a(p)$ (*e.g.*, Property 12), it can be written as $\forall s, s^0, p : T : S \cdot b[s, s^0, p] \Rightarrow preA[s^0, p]$. A property of the form "action *a* is executable only when condition *C* holds" (*e.g.*, Properties 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13) is written as: $\forall s : S \cdot preA[s, p] \Rightarrow C$. These three patterns are approximations of the property, because an action can be executed when its precondition holds and when $\exists s^0 : S \cdot postA[s, s^0, p]$ holds. In other words, the precondition must hold and the postcondition must be feasible. However, checking such an existential formula over IS states in Alloy is usually not possible, due to memory limitations. Luckily, the feasibility of postconditions is rarely an issue. Hence, we safely approximate the executability of an action to its precondition only. Invariant properties *I* for some action *a* (*e.g.*, Property 15) are easily expressed as a formula of the form $\forall s, s^0 : S, p : T \cdot I[s] \wedge a[s, s^0, p] \Rightarrow I[s^0]$. Properties expressing the reachability of a state from a condition *C* (*e.g.*, Property 14) are more tricky to express. We first tried to find a trace in the transition system, but that reveals to be impossible to check due to memory limitations. We then resorted to show the existence of a trace by describing how it can be computed. For property 14, the predicate describing such a trace states that an iteration over actions return, cancel and leave ultimately leads to a state where the member does not exist. If the property fails, it is either because the bound given for the length of the trace is too small (*i.e.*, the last state of the trace satisfies the precondition of a return, a cancel or a leave) or because the property is false in the model. By looking at the counterexample found, we can determine which case holds and increase accordingly the bounds for the trace and the number of library states. An additional difficulty is to determine the valid library states where *C* holds. These can be either characterized by a fact, which is error-prone to specify, but more efficient to check, or by executing entity and association producers from the initial state of the system to automatically construct valid library states satisfying *C*, which is simple to specify, but significantly less efficient to check.

5.6 ProB

Each action is specified as an operation defined as a precondition and a postcondition. Therefore, the main difficulty is to translate the ordering constraints (like, for example, "a book must be acquired in order to borrow it") in a precondition and find appropriate updates of state variables, as in SMV and Alloy.

As already mentioned, most of the requirement can be categorized in two patterns: (can_$A \Rightarrow P$) and ($P \Rightarrow$ can_A). In general, a requirement that looks like "*A* can be executed only if *P* is true" (the first template) can be seen as an indication that the precondition of *A* implies *P*. On the other hand, "*A* can be executed if *P* is true" (the second one) means that *P* implies the precondition of *A*. In ProB, can_*A* is expressed with the executability operator *e*(*A*). However, it denotes the exact condition under which *A* is executable; it is not an approximation as we have done in Alloy.

Specifying properties is straightforward using LTL and CTL. All properties are expressed in LTL, except 14, expressed in CTL. Property 12 is slightly more difficult to express. It denotes an ordering constraint that depends on both the current state (the book has been reserved by the member) and the previous action (once a Take is executed, the executability of the Cancel is not needed anymore). Thus, the executability operator, the *next action* operator and the LTL release operator are needed. This property does not not fit in the two described patterns.

Since ProB uses the B notation, it can be used in conjunction with Atelier B. This means that some proofs can be done prior or after using ProB. These tools can work together. For example, Property 15 is defined as an invariant. Atelier B generates proof obligations for invariants. But when the proof fails, ProB is quite useful to find where the problem is located. On the other hand, most temporal properties are generally not provable in Atelier B because they cannot be easily expressed as an invariant.

6 Analysis of the Case Study

In this section, we analyse the results of our case study along several aspects of IS specications which distinguish the salient features of each model checker.

Model specification language: abstraction over entity instances. This feature enables the specifier to parameterize the number of instances for each entity and association (*e.g.*, the number of books). If it is lacking, then the size of the specification text grows exponentially. All model checkers, except NuSMV and lotos-nt support this feature. It is worse in NuSMV, where each transition must be hardcoded for a given member and book. In lotos-nt, quantification for interleave is missing.

Model specification language: representation of entity and association structures. This is reasonably well supported by all model checkers. Modeling actions that involve several associations, like take, is not trivial in Spin.

Model specification language: representation of IS scenarios. This is also reasonably well supported by all model checkers. IS requirements are often described as scenarios on events, from which event ordering constraints are deduced. These ordering constraints are more explicitly represented in event-based languages like cadp and Spin. They are encoded as preconditions in state-based languages like Spin, NuSMV, ProB and Alloy, which are a little bit more cryptic.

Property specification language: abstraction over entity instances. Similarly, this feature enables the specifier to abstract from entity instances by using quantification on variables. If it is lacking, either the number of properties grows exponentially with the number of instances to check, or, as we did in this case study, a property is hardcoded for a particular instance of each entity, assuming that each entity behaves in a similar fashion (which may not hold in practice). NuSMV, lotos-nt and Spin lack this feature, since it is generally not supported in LTL, CTL and XTL. ProB does not suffer from this limitation in CTL and LTL, because it evaluates properties for all elements of abstract sets when necessary. Hence, only ProB, fdr2 and Alloy fully support this feature.

IS property specification. We have identified the following classes of properties for IS:

1. SCE: Sufficient state condition to enable an event (*e.g.*, case study properties 1 and 12). These are relatively easy to specify in state-based languages like NuSMV, ProB and Spin. All of these properties must be approximated in Alloy, otherwise they require a too large number of atoms to be completely checked. The validity of the approximation relies on the hypothesis that the postcondition of an action is satisfiable when the precondition holds. fdr2 can also handle these properties using stable failure refinement, but sometimes by approximation (property 1).

2. NCE: Necessary state condition to enable an event (*e.g.*, case study properties 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13). These are also relatively easy to specify in state-based languages like NuSMV, ProB and Spin, and with some approximations for Alloy (similar to SCE). Properties 5 and 10 have not been specified in XTL for cadp, because the states where too difficult to characterize using events only. There were no problem to specify these using trace refinement in fdr2. However, we suspect that there may be cases where characterizing states using only events may be difficult.

3. SCEF: Sufficient state condition to enable an event on some execution path (*e.g.*, property 14). This is easy to specify in NuSMV and ProB, thanks to CTL. It is also possible for cadp and fdr2, when the state condition is easy to characterize using events. It can be specified in Alloy by providing the path of events leading to the desired event, when such a path does not exceed the number of atoms available. It is not possible to specify these properties in Spin, since they are not supported by LTL.

4. INV: Invariant state property (*e.g.*, property 15). All model checkers can handle these without particular problems.

Property specification language: access to states and events. Since most of the properties use both states and events, model checkers that support both, like ProB and Alloy, are simpler to use, since they can represent properties more explicitly (or directly) than the others. cadp and fdr2 being event oriented, handling states is sometimes cumbersome. Spin offers limited supported for events and we have used it extensively, similarly in NuSMV, but to a lesser extent.

Execution time and number of entity instances. Figure 2 shows the execution for the number of instances (number of books and number of members). The average time per property is also provided, since not every model checker can handle all properties. Overall, cadp, NuSMV, ProB and fdr2 cannot check, within reasonable bounds of time and memory, more than 3 instances for each entity for at least one property, although for some properties they can check a few more instances. Spin can handle up to 5 entity instances. Alloy is the most efficient model checker for IS for large number of entity instances. fdr2 is the most efficient for 3 instances; it fails due to memory limitations for more than 3 instances per entity. Alloy is quite impressive: it can handle up to 98 instances for all properties except 14 in less than a minute. Property 14 is checked for 8 members and 8 books in a few minutes. With the library case study, 3 instances is a minimum to check reservation queues of length greater than 1. Note that the latest release of ProB fails for 3 properties, due to some defects which have been reported to authors. This is why we only include the results of 12 properties in Fig. 2.

Tools support. Simulators are available in each method, which is very handy to discover specification errors. The simulator in NuSMV is not straightforward to use, because it is sometimes difficult to select the transition to execute.

Step	Spin	NuSMV	fdr2	cadp	Alloy	ProB

Nb of Books/Members	3/3	5/5	3/3	3/3	3/3	3/3	8/8	3/3
Check Time	772.52	8645.6	3844.5	77.08	970.19	221.08	288.59	1094.4
Number of properties	14	14	15	15	13	15	15	12
Average (per property)	55.18	617.54	256.3	5.14	74.63	14.74	19.24	91.19

Figure 2: Model checking duration in seconds for the properties of the library specification

7 Conclusion

We have presented a comparison of six model checkers for the verification of IS. The comparison is based on a case study of a typical IS. The study reveals that a good IS model checker has to be very polyvalent. To conveniently specify IS models and properties, it should support both states and events. Process algebraic operators are desirable to easily expressed IS scenarios, while state variables are handy to streamline specification of properties. CTL seems sufficient to handle most common properties. LTL is useful, but insufficient (*e.g.*, SCEF properties). A pure first-order logic like Alloy is sufficient, but less intuitive in the case of SCEF properties. Given these characteristics, ProB seems to be the most polyvalent model checker for IS.

Since these conclusions are drawn from a single example, they must be further validated with additional examples. However, the library case study is sufficiently complex to exhibit a good number of characteristics found in most IS. It only contains two entities and two associations; large IS typically have hundred of entities and associations, but it seems quite reasonable to suppose that the verification of a property can be restricted to the entities and attributes involved. Hence, the properties checked in this case study are representative of typical IS properties.

Additional case studies would certainly find other limitations of these model checkers. For instance, our case study only addresses the sequence of actions that an IS must accept. It does not cover output delivery (*e.g.*, queries) and user interface interactions.

References

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

[2] Augusto, J.C., Ferreira, C., Gravell, A.M., Leuschel, M., Ng, K.M.Y.: The benefits of rapid modelling for e-business system development. ER Workshops pp. 17–28 (2003)

[3] Aydal, E.G., Utting, M., Woodcock, J.: A comparison of state-based modelling tools for model validation. TOOLS-Europe08, Switzerland (2008)

[4] Biere, A., Clarke, E.M., Cimatti, A., Zhu, Y.: Symbolic model checking without BDDs. In: Proceeding of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99). LNCS, vol. 1579, pp. 193–207 (1999)

[5] Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. In: van Eijk,

P.H.J., Vissers, C.A., Diaz, M. (eds.) The Formal Description Technique LOTOS, pp. 23–73. Elsevier Science Publishers B.V. (1989)

[6] Chane-Yack-Fa, R., Fraikin, B., Frappier, M., Chossard, R., Ouenzar, M.: Comparison of model checking tools for information systems. Tech. Rep. 29, Université de Sherbrooke (Jun 2010), disponible from http://pages.usherbrooke.ca/gril/TR/TR-GRIL-1006-29.pdf

[7] Clarke, E.M., Emerson, E.A.: Synthesis of synchronization skeletons for branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs Workshop. LNCS, vol. 131. Springer-Verlag (1981)

[8] Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven web applications. Journal of Computer and System Sciences 73(3), 442–474 (2007)

[9] Emerson, E.A., Halpern, J.Y.: “Sometimes” and “Not Never” revisited: On branching versus linear time temporal logic. J. ACM 33(1), 151–178 (1986)

[10] Garavel, H.: Compilation et vérification de programmes LOTOS. Ph.D. thesis, Université Joseph Fourier, Grenoble (November 1989)

[11] Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)

[12] Jackson, D.: Software Abstractions. MIT Press (2006)

[13] Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: Formal Methods. LNCS, vol. 2805, pp. 855–874. Springer-Verlag (2003)

[14] Mateescu, R., Garavel, H.: XTL: A meta-language and tool for temporal logic model-checking. In: Proceedings of the International Workshop on Software Tools for Technology Transfer STTT’98. p. 10. Aalborg, Denmark (Jul 1998)

[15] McMillan, K.L.: Symbolic Model Checking. Ph.D. thesis, Carnegie Mellon University (1993)

[16] Morimoto, S.: A survey of formal verification for business process modeling. Lecture Notes in Computer Science 5102, 514–524 (2008)

[17] Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science, 18th Annual Symposium on. pp. 46–57 (1977)

[18] Roscoe, B.A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR, amended 2005, 3rd edn. (1998)

[19] Spielmann, M.: Abstract state machines: Verification problems and complexity. Ph.D. thesis, Bibliothek der RWTH Aachen (2000)

[20] Yeung, W.L., Leung, K.R.P.H., Wang, J., Dong, W.: Modelling and model checking suspendible business processes via statechart diagrams and CSP. Science of Computer Programming 65(1), 14–29 (2007)

A Spin

```
inline pre_join(mId) { true } inline post_join(mId)
{ is_member[mId]=true
}

inline pre_leave(mId)
{
      ( !nb_loans[mId] &&
         !is_reserving_n_books[mId] )
} inline post_leave(mId)
{ is_member[mId]=false ; goto out }

inline pre_acquire (bId) { true } inline post_acquire (bId)
{ is_acquired[bId]=true
}

inline pre_discard (bId)
{
      ( !is_borrowed[bId] && empty(is_reserved_by[bId]) )
} inline post_discard (bId)
{ is_acquired[bId]=false ; goto
      not_acquired
}
```

Listing 1: Macro of the pre-post condition of the entities actions in the Promela specification

```
inline pre_lend (mId, bId)
{
      ( is_member[mId] && is_acquired[bId] &&
         !is_borrowed[bId] && (nb_loans[mId] <
         NBMAXLOANS) && empty(is_reserved_by[bId]) )
} inline post_lend (mId, bId)
{ nb_loans[mId]++ ; is_borrowed[bId] =
      true ; is_borrowed_by[bId] = mId
} inline pre_renew (mId, bId)
{
      ( empty(is_reserved_by[bId]) )
} inline post_renew (mId, bId)
{ goto borrowed
}
```

Listing 2: Macro of the pre-post condition of the association actions in the Promela specification

```
inline pre_return (mId, bId) { true } inline post_return (mId,
bId)
{ is_borrowed[bId] = false ; nb_loans[mId]-
      - ; goto not_borrowed
}
```

```
inline pre_reserve (mId, bId)
{
      ( is_member[mId] && is_acquired[bId] &&
         ( is_borrowed[bId] || nempty(is_reserved_by[bId]) ) &&
         ( !is_borrowed[bId] || (is_borrowed_by[bId] != mId) ) &&
                        ( empty(is_reserved_by[bId]) || !is_reserved_by[bId]??[eval(mId)] ))
} inline post_reserve (mId, bId)
{ is_reserved_by[bId]!mId ;
      is_reserving_n_books[mId]++
} inline pre_cancel (mId, bId) { true } inline
post_cancel (mId, bId)
{ is_reserved_by[bId]??eval(mId) ;
      is_reserving_n_books[mId]-- ; goto
      not_reserved
}
```

Listing 3: Macro of the pre-post condition of the entities actions in the Promela specification (part 2)

```
inline pre_take (mId,bId)
{ pre_take_in_loan(mId,bId) &&
      pre_take_in_reservation(mId,bId)
}

inline pre_take_in_loan (mId,bId) { true } inline post_take_in_loan
(mId, bId)
{ nb_loans[mId]++ ; is_borrowed[bId] =
      true ; is_borrowed_by[bId] = mId
}

inline pre_take_in_reservation (mId, bId)
{
      ( is_member[mId] && is_acquired[bId] &&
         !is_borrowed[bId] && (nb_loans[mId] <
         NBMAXLOANS) &&
         is_reserved_by[bId]?[eval(mId)] )
} inline post_take_in_reservation (mId, bId)
{ is_reserved_by[bId]?eval(mId) ;
      is_reserving_n_books[mId]-- ; goto
      not_reserved
}

inline pre_list_of_books_and_borrower () { true } inline
post_list_of_books_and_borrower ()
{ goto progress_system_running
}
```

Listing 4: Macro of the pre-post condition of the take action in the Promela specification

```
int nb_loans [NB_MEMBER] ; bool is_acquired
[NB_BOOK] ; bool is_member [NB_MEMBER] ;
bool is_borrowed [NB_BOOK] ; MEMBERID
is_borrowed_by [NB_BOOK] ;
chan is_reserved_by [NB_BOOK] = [NB_MEMBER] of { MEMBERID } ; int
is_reserving_n_books[NB_MEMBER] ;

mtype = { from_reservation, from_loan } ; chan ready_to_take[NB_BOOK] = [0] of {
mtype, MEMBERID } ;
```

Listing 5: Global variable of the Promela specification

```
proctype member(MEMBERID mId)
{ out: left: progress_member: end_member:
        atomic{ /*pre_join(mId) ->*/ printf("join(%d)\n",mId) ;
                post_join(mId)
                } ;

in: joined:
        atomic{ pre_leave(mId) ->
                printf("leave(%d)\n",mId) ;
                post_leave(mId) }
}
```

Listing 6: Definition of the member process in the Promela specification

```
proctype book(BOOKID bId)
{ not_acquired: discarded: progress_book: end_book:
        atomic{ /*pre_acquire(bId) ->*/ printf("acquire(%d)\n", bId) ;
                post_acquire(bId)
                } ;

acquired:
        atomic{ pre_discard(bId) -> printf("discard(%d)\n",
                bId) ; post_discard(bId) }
}
```

Listing 7: Definition of the book process in the Promela specification

```
proctype loan(MEMBERID mId; BOOKID bId)
{ not_borrowed: end_loan: progress_loan: if
```

```
    :: atomic{ pre_lend(mId, bId) -> printf("lend(%d,%d)\n", mId, bId) ;
                post_lend(mId, bId)
                }
    :: atomic{ ready_to_take[bId]?from_reservation,eval(mId) ->
                /* pre_take_in_loan(mId, bId) -> */ printf("take
    awaiting(%d,%d)\n", mId, bId) ; post_take_in_loan(mId, bId) ;
    ready_to_take[bId]!from_loan,mId } fi ;

borrowed: if
    :: atomic{ pre_renew(mId, bId) -> printf("renew(%d,%d)\n", mId, bId) ;
                post_renew(mId, bId)
                }
    :: atomic{ /*pre_return(mId, bId) ->*/ printf("return(%d,%d)\n", mId, bId) ;
                post_return(mId, bId)
                } fi
}
```

Listing 8: Definition of the loan process in the Promela specification

```
proctype reservation(MEMBERID mId; BOOKID bId)
{ not_reserved: end_reservation: progress_reservation:
    if
    :: atomic{ pre_reserve(mId, bId) -> printf("reserve(%d,%d)\n", mId, bId) ;
                post_reserve(mId, bId) ;
                } fi ;

reserved: if
    :: atomic{ pre_take_in_reservation(mId, bId) -> ready_to_take[bId]!from_reservation,mId
                } ;
       is_taking:
       atomic{ ready_to_take[bId]?from_loan,eval(mId) ; printf("take(%d,%d)\n", mId, bId) ;
                post_take_in_reservation(mId, bId)
                }
    :: atomic{ /*pre_cancel(mId, bId) ->*/ printf("cancel(%d,%d)\n", mId, bId) ;
                post_cancel(mId, bId)
                } fi
}
```

Listing 9: Definition of the reservation process in the Promela specification

1. A book can always be acquired by the library when it is not currently acquired.
   `<>b0_can_be_acquired && [](b0_is_discarded -> b0_can_be_acquired)`
2. A book cannot be acquired by the library if it is already acquired.
   `[] (!(b0_is_acquired && b0_is_discarded))`
3. An acquired book can be discarded only if it is neither lent nor reserved.
   `[] (b0_is_discarded -> b0_is_neither_borrowed_nor_reserved)`
4. A person must be a member of the library in order to borrow a book.
   `[] (b0_can_be_borrowed_by_m0 -> m0_has_joined)`
5. A book can be reserved only if it has been borrowed or already reserved by another member.

[] (b0_can_be_reserved_by_m0 -> (b0_is_borrowed || (!b0_is_not_reserved && !b0_is_reserved_by_m0))

6. A book cannot be reserved by the member who is borrowing it.
   [] (b0_can_be_reserved_by_m0 -> !b0_is_borrowed_by_m0)
7. A book cannot be reserved by the member who is reserving it.
   [] (b0_can_be_reserved_by_m0 -> !b0_is_reserved_by_m0)
8. A book cannot be lent to a member if it is reserved.
   [] (b0_is_reserved -> !b0_can_be_lent_to_m0)
9. A member cannot renew a loan if the book is reserved.
   [] (b0_is_reserved -> !m0_can_renew_loan_b0)
10. A member is allowed to take a reserved book only if he owns the oldest reservation.
    [] (b0_can_be_taken_by_m0 -> b0_is_reserved_by_m0_first)
11. A book can be taken only if it is not borrowed.
    [] (b0_can_be_taken_by_m0 -> !b0_is_borrowed)
12. Anyone who has reserved a book can cancel the reservation at anytime before he takes it.
    [](b0_is_reserved_by_m0 -> ((m0_can_cancel_b0_reservation U m0_is_taking_b0_reservation)||
    []!m0_is_taking_b0_reservation)
13. A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or canceled.
    [](m0_can_leave -> (m0_has_no_reservation && m0_has_no_loan)
14. Ultimately, there is always a procedure that enables a member to leave the library. This is impossible in LTL.
15. A member cannot borrow more than the loan limit defined at the system level for all users.
    [] (nb_of_loan_of_m0_is_less_than_max)

Figure 3: Requirements properties expressed in LTL for Spin model checker (part 2)

```
#define b0_is_acquired (book[b_pid[0]]@acquired)
#define b0_is_discarded (book[b_pid[0]]@discarded)
#define b0_is_borrowed (is_borrowed[0])

#define b0_is_not_reserved (empty(is_reserved_by[0]))
#define b0_is_reserved (nempty(is_reserved_by[0]))

#define b0_is_neither_borrowed_nor_reserved ( !b0_is_borrowed && b0_is_not_reserved )
#define b0_is_either_borrowed_or_reserved ( b0_is_borrowed || b0_is_reserved )

#define m0_has_joined (member[m_pid[0]]@joined)
#define nb_of_loan_of_m0_is_less_than_max (nb_loans[0] <= NBMAXLOANS)
#define nb_of_loan_of_m0_is_strictly_less_than_max (nb_loans[0] < NBMAXLOANS)
#define m0_has_no_reservation (!is_reserving_n_books[0])
#define m0_has_no_loan (!nb_loans[0])

#define b0_is_borrowed_by_m0 (loan[l_pid[0]]@borrowed)
#define b0_is_not_borrowed_by_m0 (loan[l_pid[0]]@not_borrowed)

#define b0_is_reserved_by_m0 (reservation[r_pid[0]]@reserved)
```

```
#define m0_is_taking_b0_reservation (reservation[r_pid[0]]@is_taking)

#define b0_is_reserved_by_m0_first (is_reserved_by[0]?[0])
```

Listing 10: Predicate definition in Spin

```
#define b0_can_be_acquired ( b0_is_discarded )
#define b0_can_be_lent_to_m0 ( b0_is_not_borrowed_by_m0 && m0_has_joined &&
                               b0_is_acquired && !b0_is_borrowed &&
                               nb_of_loan_of_m0_is_strictly_less_than_max &&
                               b0_is_not_reserved )
#define b0_can_be_taken_by_m0 ( b0_is_not_borrowed_by_m0 && m0_has_joined &&
                                b0_is_acquired && !b0_is_borrowed &&
                                nb_of_loan_of_m0_is_strictly_less_than_max &&
                                b0_is_reserved_by_m0_first )
#define b0_can_be_borrowed_by_m0 ( b0_is_not_borrowed_by_m0 &&
                                   (b0_can_be_lent_to_m0 || b0_can_be_taken_by_m0) )
#define b0_can_be_reserved_by_m0 ( b0_is_not_reserved_by_m0 &&
                                   m0_has_joined && b0_is_acquired && ( b0_is_borrowed
                                   || b0_is_reserved ) &&
                                      ( !b0_is_borrowed || !b0_is_borrowed_by_m0 ) &&
                                         ( b0_is_not_reserved || !b0_is_reserved_by_m0 ) )
#define m0_can_renew_loan_b0 ( b0_is_borrowed_by_m0 && b0_is_not_reserved )
#define m0_can_cancel_b0_reservation ( b0_is_reserved_by_m0 )
#define m0_can_leave ( m0_has_joined && m0_has_no_loan &&
                       m0_has_no_reservation )
```

Listing 11: “can_xxx” predicate definition in Spin

B NuSMV

```
MODULE main
VAR member1 : member(1); member2 : member(2); member3 :
  member(3); join1 : process join(member1); join2 : process
  join(member2); join3 : process join(member3); leave1 : process
  leave(member1); leave2 : process leave(member2); leave3 :
  process leave(member3); book1 : book(1); book2 : book(2);
  book3 : book(3); acquire1 : process acquire(book1); acquire2 :
  process acquire(book2); acquire3 : process acquire(book3);
  discard1 :process discard(book1); discard2 :process
  discard(book2); discard3 :process discard(book3); lend1_1 :
  process lend(member1,book1); lend1_2 : process
  lend(member1,book2); lend1_3 : process
  lend(member1,book3); lend2_1 : process
  lend(member2,book1); lend2_2 : process
  lend(member2,book2); lend2_3 : process
  lend(member2,book3); lend3_1 : process
  lend(member3,book1); lend3_2 : process
  lend(member3,book2); lend3_3 : process
  lend(member3,book3); return1_1 : process
  return(member1,book1); renew1_1 : process
  renew(member1,book1); reserve1_1 : process
  reserve(member1,book1); take1_1 : process
  take(member1,book1); cancel1_1 : process
  cancel(member1,book1); -- etc.
```

Listing 12: Main Module in SMV

```
MODULE member(memberId)
VAR mId : 0..3; is_member : boolean;
  nb_loans : 0..2; is_reserving_n_books :
  0..3;
INVAR mId = memberId;
ASSIGN init(mId) := memberId; init(is_member)
  := 0; init(nb_loans) := 0;
  init(is_reserving_n_books) := 0;
```

Listing 13: Member Module in SMV

```
MODULE book(bookId)
DEFINE no_reservation := is_reserved_by[0] = 0;
VAR bId : 0..3; is_acquired : boolean; is_borrowed :
  boolean; is_borrowed_by : 0..3; is_reserved_by :
  array 0..3 of 0..3;
INVAR bId = bookId;
ASSIGN init(bId) := bookId;
  init(is_reserved_by[0]) := 0;
  init(is_reserved_by[1]) := 0;
  init(is_reserved_by[2]) := 0;
  init(is_reserved_by[3]) := 0;
  init(is_acquired) := 0; init(is_borrowed) :=
  0; init(is_borrowed_by) := 0;
```

Listing 14: Book Module in SMV

```
MODULE join(member)
DEFINE pre_join := !member.is_member;
IVAR do : boolean;
ASSIGN next(member.is_member) :=
    case
      pre_join : 1;
      1 : member.is_member; esac;
TRANS do <-> (running & pre_join);
```

Listing 15: Join Module in SMV

```
MODULE leave(member)
DEFINE pre_leave := member.is_member
    & member.nb_loans = 0
```

```
    & member.is_reserving_n_books = 0;
IVAR do : boolean;
ASSIGN next(member.is_member) :=
    case
      pre_leave : 0;
      1 : member.is_member; esac;
TRANS do <-> (running & pre_leave);
```

Listing 16: Leave Module in SMV

```
MODULE acquire(book)
DEFINE pre_acquire := !book.is_acquired;
IVAR do : boolean;
ASSIGN next(book.is_acquired) :=
    case
      pre_acquire : 1;
      1 : book.is_acquired; esac;
TRANS (running & pre_acquire) <-> do;
```

Listing 17: Acquire Module in SMV

```
MODULE discard(book)
DEFINE pre_discard := book.is_acquired
    & !book.is_borrowed
    & book.no_reservation;
IVAR do : boolean;
ASSIGN next(book.is_acquired) :=
    case
      pre_discard : 0;
      1 : book.is_acquired; esac;
TRANS (running & pre_discard) <-> do;
```

Listing 18: Discard Module in SMV

```
MODULE lend(member,book)
DEFINE pre_lend := member.is_member & book.is_acquired
    & !book.is_borrowed & book.no_reservation
    & member.nb_loans < 2; -- upper bound and max loans allowed
IVAR do : boolean;
ASSIGN next(book.is_borrowed) :=
    case
      pre_lend : 1;
      1 : book.is_borrowed; esac;
  next(member.nb_loans) :=
```

```
    case
      pre_lend : member.nb_loans + 1;
      1 : member.nb_loans; esac;
  next(book.is_borrowed_by) := case
      pre_lend : member.mId;
      1 : book.is_borrowed_by; esac;
TRANS do <-> (running & pre_lend);
```

Listing 19: Lend Module in SMV

```
MODULE return(member,book)
DEFINE pre_return := book.is_borrowed & book.is_borrowed_by = member.mId
    & member.nb_loans > 0; -- lower bound
IVAR do : boolean;
ASSIGN next(book.is_borrowed) :=
    case
      pre_return : 0;
      1 : book.is_borrowed; esac;
  next(member.nb_loans) :=
    case
      pre_return : member.nb_loans - 1;
      1 : member.nb_loans; esac;
  next(book.is_borrowed_by) :=
    case
      pre_return : 0;
      1 : book.is_borrowed_by; esac;
TRANS do <-> (running & pre_return);
```

Listing 20: Return Module in SMV

```
MODULE renew(member,book)
DEFINE pre_renew := book.is_borrowed_by = member.mId & book.no_reservation;
IVAR do : boolean;
ASSIGN next(book.is_borrowed) :=
    case
      pre_renew : 1;
      1 : book.is_borrowed; esac;
  next(member.nb_loans) :=
    case
      pre_renew : member.nb_loans;
      1 : member.nb_loans; esac;
  next(book.is_borrowed_by) := case
      pre_renew : member.mId;
      1 : book.is_borrowed_by; esac;
TRANS do <-> (running & pre_renew);
```

Listing 21: Renew Module in SMV

```
MODULE reserve(member,book)
DEFINE already_reserved :=
```

```
      book.is_reserved_by[0] = member.mId |
      book.is_reserved_by[1] = member.mId
      | book.is_reserved_by[2] = member.mId
      | book.is_reserved_by[3] = member.mId; pre_reserve :=
      member.is_member & book.is_acquired
      & !already_reserved
      & ((book.is_borrowed & book.is_borrowed_by != member.mId)
                  | (!book.is_borrowed & book.is_reserved_by[0]>0))
      & book.is_reserved_by[3] = 0
      & member.is_reserving_n_books<3; -- upper bound constraint
IVAR do : boolean;
ASSIGN next(book.is_reserved_by[0]) :=
    case
      pre_reserve & book.is_reserved_by[0]=0 : member.mId;
      1 : book.is_reserved_by[0]; esac;
  next(book.is_reserved_by[1]) :=
    case
      pre_reserve & book.is_reserved_by[0]>0 &
      book.is_reserved_by[1]=0 : member.mId;
      1 : book.is_reserved_by[1]; esac;
  next(book.is_reserved_by[2]) :=
    case
      pre_reserve & book.is_reserved_by[1]>0 &
      book.is_reserved_by[2]=0 : member.mId;
      1 : book.is_reserved_by[2]; esac;
  next(book.is_reserved_by[3]) :=
    case
      pre_reserve & book.is_reserved_by[2]>0 : member.mId;
      1 : book.is_reserved_by[3]; esac;
  next(member.is_reserving_n_books) :=
    case
      pre_reserve : (member.is_reserving_n_books + 1);
      1 : member.is_reserving_n_books; esac;
TRANS do <-> (running & pre_reserve);
```

Listing 22: Reserve Module in SMV

```
MODULE take(member,book)
DEFINE pre_take := !book.is_borrowed & member.nb_loans < 2
      & member.mId = book.is_reserved_by[0]
      & member.is_reserving_n_books > 0; -- lower bound constraint
IVAR do : boolean;
ASSIGN next(book.is_reserved_by[0]) :=
    case
      pre_take : book.is_reserved_by[1];
      1 : book.is_reserved_by[0]; esac;
  next(book.is_reserved_by[1]) :=
    case
      pre_take : book.is_reserved_by[2];
```

```
        1 : book.is_reserved_by[1]; esac;
    next(book.is_reserved_by[2]) :=
      case
        pre_take : book.is_reserved_by[3];
        1 : book.is_reserved_by[2]; esac;
    next(book.is_reserved_by[3]) :=
      case
        pre_take : 0;
        1 : book.is_reserved_by[3]; esac;
    next(member.is_reserving_n_books) :=
      case
        pre_take : (member.is_reserving_n_books - 1);
        1 : member.is_reserving_n_books; esac;
      -- lend postconditions next(book.is_borrowed) :=
      case
        pre_take : 1;
        1 : book.is_borrowed; esac;
    next(member.nb_loans) :=
      case
        pre_take : member.nb_loans + 1;
        1 : member.nb_loans; esac;
    next(book.is_borrowed_by) := case
        pre_take : member.mId;
        1 : book.is_borrowed_by; esac;
TRANS do <-> (running & pre_take);
```

Listing 23: Take Module in SMV

```
MODULE cancel(member,book)
DEFINE first := member.mId = book.is_reserved_by[0]; second := member.mId =
  book.is_reserved_by[1]; third := member.mId = book.is_reserved_by[2]; fourth :=
  member.mId = book.is_reserved_by[3]; pre_cancel := (first | second | third | fourth) &
  member.is_reserving_n_books > 0; -- lower bound constraint
IVAR do : boolean;
ASSIGN
  -- fifo postconditions next(book.is_reserved_by[0]) :=
    case
      pre_cancel & first : book.is_reserved_by[1];
      1 : book.is_reserved_by[0]; esac;
  next(book.is_reserved_by[1]) :=
    case
      pre_cancel & (first | second) : book.is_reserved_by[2];
      1 : book.is_reserved_by[1]; esac;
  next(book.is_reserved_by[2]) :=
    case
      pre_cancel & (first | second | third) : book.is_reserved_by[3];
      1 : book.is_reserved_by[2]; esac;
  next(book.is_reserved_by[3]) :=
```

```
    case
      pre_cancel & (first | second | third | fourth) : 0;
      1 : book.is_reserved_by[3]; esac;
  next(member.is_reserving_n_books) :=
    case
      pre_cancel : (member.is_reserving_n_books - 1);
      1 : member.is_reserving_n_books; esac;
TRANS do <-> (running & pre_cancel);
```

Listing 24: Cancel Module in SMV

```
-- [1] A book can always be acquired by the library when it is not currently acquired.
SPEC AG (!book1.is_acquired -> EX book1.is_acquired);
INVARSPEC (!book1.is_acquired -> acquire1.pre_acquire);

-- [2] A book cannot be acquired by the library if it is already acquired.
INVARSPEC (book1.is_acquired -> !acquire1.pre_acquire);
LTLSPEC G (acquire1.do -> X((!acquire1.do U discard1.do) | G !acquire1.do));

-- [3] An acquired book can be discarded only if it is neither lent nor reserved.
LTLSPEC (G (lend1_1.do -> X((!discard1.do U return1_1.do) | G !discard1.do))) & (G (reserve1_1.do ->
  X((!discard1.do U (cancel1_1.do | take1_1.do)) | G ! discard1.do)))
  & (G (take1_1.do -> X((!discard1.do U return1_1.do) | G !discard1.do)));
LTLSPEC G ((book1.is_borrowed | !book1.no_reservation) -> X book1.is_acquired);

-- [4] A person must be a member of the library in order to borrow a book.
LTLSPEC G (lend1_1.do -> ((!leave1.do) S join1.do));
INVARSPEC (!member1.is_member -> member1.nb_loans = 0);
INVARSPEC (lend1_1.pre_lend -> member1.is_member);

-- [5] A book can be reserved only if it has been borrowed or already reserved by another member.
INVARSPEC (reserve1_1.pre_reserve -> (book1.is_borrowed | book1.is_reserved_by [0]>0));
LTLSPEC G (reserve1_1.do -> ((!return2_1.do S (lend2_1.do | take2_1.do))
            | (!return3_1.do S (lend3_1.do | take3_1.do))
            | ((!cancel2_1.do & !take2_1.do) S reserve2_1.do)
            | ((!cancel3_1.do & !take3_1.do) S reserve3_1.do)));

-- [6] A book cannot be reserved by the member who is borrowing it.
INVARSPEC (book1.is_borrowed_by = member1.mId -> !reserve1_1.pre_reserve);
LTLSPEC G (reserve1_1.do ->
      ((!(lend1_1.do | take1_1.do) S return1_1.do) | H !(lend1_1.do | take1_1.do) ));
INVARSPEC (book1.is_borrowed_by = member1.mId -> (book1.is_reserved_by[0]!= member1.mId
                & book1.is_reserved_by[1]!=member1.mId
                & book1.is_reserved_by[2]!=member1.mId &
                book1.is_reserved_by[3]!=member1.mId));
```

Listing 25: Properties in SMV (part 1)

```
-- [7] A book cannot be reserved by a member who is reserving it. INVARSPEC
((book1.is_reserved_by[0]=member1.mId
        | book1.is_reserved_by[1]=member1.mId
        | book1.is_reserved_by[2]=member1.mId
        | book1.is_reserved_by[3]=member1.mId)
        -> !reserve1_1.pre_reserve);
LTLSPEC G (reserve1_1.do ->
      Y ((!reserve1_1.do S (cancel1_1.do | take1_1.do)) | H !reserve1_1.do)); INVARSPEC
(book1.is_reserved_by[0]=book1.is_reserved_by[1] -> book1. is_reserved_by[0]=0
         & book1.is_reserved_by[0]=book1.is_reserved_by[2] -> book1.is_reserved_by
              [0]=0
         & book1.is_reserved_by[0]=book1.is_reserved_by[3] -> book1.is_reserved_by
              [0]=0
         & book1.is_reserved_by[1]=book1.is_reserved_by[2] -> book1.is_reserved_by
              [1]=0
         & book1.is_reserved_by[1]=book1.is_reserved_by[3] -> book1.is_reserved_by
              [1]=0
         & book1.is_reserved_by[2]=book1.is_reserved_by[3] -> book1.is_reserved_by [2]=0);

-- [8] A book cannot be lent to a member if it is reserved.
INVARSPEC (lend1_1.pre_lend -> book1.no_reservation);
LTLSPEC G (lend1_1.do -> (((!reserve1_1.do S (cancel1_1.do | take1_1.do)) | H ! reserve1_1.do)
          & ((!reserve2_1.do S (cancel2_1.do | take2_1.do)) | H !reserve2_1.do)) & ((!reserve3_1.do S
          (cancel3_1.do | take3_1.do)) | H !reserve3_1.do));

-- [9] A member cannot renew a loan if the book is reserved.
INVARSPEC (renew1_1.pre_renew -> book1.no_reservation);
LTLSPEC G (renew1_1.do -> (((!reserve1_1.do S (cancel1_1.do | take1_1.do)) | H
      !reserve1_1.do)
                                          & ((!reserve2_1.do S (cancel2_1.do | take2_1.do)) | H !reserve2_1.do))
                                           & ((!reserve3_1.do S (cancel3_1.do | take3_1.do)) | H !reserve3_1.do));

-- [10] A member is allowed to take a reserved book only if he owns the oldest reservation.
INVARSPEC (take1_1.pre_take -> book1.is_reserved_by[0]=member1.mId);

-- [11] A book can be taken only if it is not borrowed.
INVARSPEC (take1_1.pre_take -> !book1.is_borrowed);
LTLSPEC G (take1_1.do ->
      Y (((!(take1_1.do | lend1_1.do) S return1_1.do) | H !(take1_1.do | lend1_1. do))
                                          & ((!(take2_1.do | lend2_1.do) S return2_1.do) | H !(take2_1.do | lend2_1
            .do))
                                          & ((!(take3_1.do | lend3_1.do) S return3_1.do) | H !(take3_1.do | lend3_1
            .do))));
```

Listing 26: Properties in SMV (part 2)

```
-- [12] Anyone who has reserved a book can cancel the reservation at anytime before he takes it.
```

```
SPEC AG (book1.is_reserved_by[0]=member1.mId ->
    EX (book1.is_reserved_by[0]!=member1.mId & book1.is_borrowed_by!=member1.mId)
    & book1.is_reserved_by[1]=member1.mId ->
    EX (book1.is_reserved_by[1]!=member1.mId & book1.is_reserved_by[0]!=member1. mId)
    & book1.is_reserved_by[2]=member1.mId ->
    EX (book1.is_reserved_by[2]!=member1.mId & book1.is_reserved_by[1]!=member1. mId)
    & book1.is_reserved_by[3]=member1.mId ->
    EX (book1.is_reserved_by[3]!=member1.mId & book1.is_reserved_by[2]!=member1. mId));
INVARSPEC ((book1.is_reserved_by[0]=member1.mId
        | book1.is_reserved_by[1]=member1.mId
        | book1.is_reserved_by[2]=member1.mId
        | book1.is_reserved_by[3]=member1.mId)
        -> cancel1_1.pre_cancel);

-- [13] A member can relinquish library membership only when all his loans have been returned and all his
        reservations have either been used or canceled.
LTLSPEC G (leave1.do -> ((!(lend1_1.do|take1_1.do) S return1_1.do | H !(lend1_1
      .do|take1_1.do))
        & (!(lend1_2.do|take1_2.do) S return1_2.do | H !(lend1_2.do|take1_2.do))
        & (!(lend1_3.do|take1_3.do) S return1_3.do | H !(lend1_3.do|take1_3.do))
        & (!reserve1_1.do S (cancel1_1.do|take1_1.do) | H !reserve1_1.do)
        & (!reserve1_2.do S (cancel1_2.do|take1_2.do) | H !reserve1_2.do)
        & (!reserve1_3.do S (cancel1_3.do|take1_3.do) | H !reserve1_3.do))); INVARSPEC (!member1.is_member ->
(member1.nb_loans=0 & member1. is_reserving_n_books=0));

-- [14] ultimately, there is always a procedure that enables a member to leave the library
SPEC AG (member1.is_member -> EF (!member1.is_member));

-- [15] A member cannot borrow more than the loan limit defined at the system level for all users.
INVARSPEC (member1.nb_loans <= 2);
```

Listing 27: Properties in SMV (part 3)

C fdr2

```
include "utility.csp"
nbMembers = 3 nbBooks = 3
MEMBERID = {1..nbMembers}
BOOKID = {1..nbBooks} nullmid = 0
maxNbLoans = nbBooks-1

channel join, leave : MEMBERID channel acquire, discard, return, renew :
BOOKID channel lend, take, cancel, reserve : MEMBERID . BOOKID

BSET = {| acquire,discard,lend,renew,return,reserve,take,cancel |}
MSET = {| join,leave,lend,renew,return,reserve,take,cancel |}
LOANSET = {| lend,renew,return,take |}
```

```
RESERVSET = {| reserve,take,cancel |}
BOOKMEMBERASSOC = union(LOANSET,RESERVSET)
BOOKCTRLSET = {| reserve, take, cancel, lend, renew,return |}
MEMBERCTRLSET = {| join,leave,lend,return,take |}

MEMBER(m) = join.m ->
      (
        (||| b : BOOKID @ LOANMEMBER(m,b))
      [| {| take |} |]
        (||| b : BOOKID @ RESERVATION(m,b))
      ); leave.m ->
      MEMBER(m)

LOANMEMBER(m,b) = LOANMEMBERCYCLE(m,b) [] SKIP

LOANMEMBERCYCLE(m,b) = (lend.m.b -> BORROWEDMEMBER(m,b)) [] (take.m.b -> BORROWEDMEMBER(m,b))

BORROWEDMEMBER(m,b) = (renew.b -> BORROWEDMEMBER(m,b)) [] (return.b -> LOANMEMBER(m,b))

MEMBERCTRL(m,s) =
            ( join!m -> MEMBERCTRL(m,{}) )
      [] ( leave!m -> MEMBERCTRL(m,{}) )
      [] (card(s) < maxNbLoans & lend!m?b -> MEMBERCTRL(m,union(s,{b})) )
      [] (card(s) < maxNbLoans & take!m?b -> MEMBERCTRL(m,union(s,{b})) )
      [] ([] b:BOOKID @ member(b,s) & return!b -> MEMBERCTRL(m,diff(s,{b})) )
```

Listing 28: Library specification in fdr2 (part 1)

```
BOOK(b) = acquire.b ->
      (
        (LOANBOOK(b))
      [| {| take |} |]
        (||| m : MEMBERID @ RESERVATION(m,b))
      ); discard.b ->
      BOOK(b)

LOANBOOK(b) = ([] m : MEMBERID @ LOANBOOKCYCLE(m,b)) [] SKIP

LOANBOOKCYCLE(m,b) =
      ( lend.m.b ->
        KLEENE_RENEW(m,b); true &
        return.b -> LOANBOOK(b)
      )
   []
      ( take.m.b ->
        KLEENE_RENEW(m,b);
        return.b ->
        LOANBOOK(b)
      )
```

```
KLEENE_RENEW(m,b) = (renew.b -> KLEENE_RENEW(m,b)) [] SKIP

RESERVATION(m,b) = (RESERVATIONCYCLE(m,b)) [] SKIP

RESERVATIONCYCLE(m,b) = reserve.m.b ->
    (
      (true & take.m.b -> RESERVATION(m,b))
    []
      (true & cancel.m.b -> RESERVATION(m,b))
    )
```

Listing 29: Library specification in fdr2 (part 2)

```
BOOKCTRL(b,l,cb) =
        ([] m:MEMBERID @ cb != m and
                                ( cb != nullmid
                                or
                                  l != <>
                                ) and
                                length(l) < nbMembers -- necessary to bound the length of the sequence; otherwise
                                    FDR2 does not terminate
                                                                & reserve!m!b -> BOOKCTRL(b,l^<m>,cb))
    [] ([] m:MEMBERID @ l != <> and m == head(l) & take!m!b -> BOOKCTRL(b,tail(l ),m))
    [] ([] m:MEMBERID @ elem(m,l) & cancel!m!b -> BOOKCTRL(b,removex(l,m),cb))
    [] ( l == <> & lend?m!b -> BOOKCTRL(b,l,m))
    [] ( l == <> & renew!b -> BOOKCTRL(b,l,cb))
    [] true & return!b -> BOOKCTRL(b,l,nullmid)

ALLBOOKS = ||| b : BOOKID @ BOOK(b)

ALLMEMBERS = ||| m : MEMBERID @ MEMBER(m)

ALLBOOKSCTRL = ||| b : BOOKID @ BOOKCTRL(b,<>,nullmid)

ALLMEMBERSCTRL = ||| m : MEMBERID @ MEMBERCTRL(m,{})

MAIN =
      (
        (ALLBOOKS [| BOOKMEMBERASSOC |] ALLMEMBERS)
      [| BOOKCTRLSET |] ALLBOOKSCTRL
      )
    [| MEMBERCTRLSET |] ALLMEMBERSCTRL
```

Listing 30: Library specification in fdr2 (part 3)

```
include "biblio.csp"
-- Property 1 : A book can always be acquired by the library when it is not currently acquired.

Prop1v0 = (|||b:BOOKID @ Prop1v0Body(b)) ||| CHAOS({|join,leave|})
Prop1v0Body(b) = acquire.b ->
    discard.b -> Prop1v0Body(b)
```

```
-- Can't check this property when both associations (loans and reservations) are active.
-- Hiding and loops in one association make all states unstable in the other association,
-- thus weakening the property to check under stable-failure refinement.
-- Must check it in two parts:

-- Part 1 - disable reservations, by synchronizing with STOP on reservation actions
assert Prop1v0 [F= (MAIN [| RESERVSET |] STOP) \ BOOKMEMBERASSOC

-- Part 2 - disable loans, by synchronizing with STOP on loan actions assert Prop1v0 [F= (MAIN [| LOANSET |]
STOP) \ BOOKMEMBERASSOC

-- Property 2. A book cannot be acquired by the library if it is already acquired.

Prop2v0 = |||b:BOOKID @ Prop2v0Body(b)
Prop2v0Body(b) = acquire.b ->
      discard.b ->
      Prop2v0Body(b)

assert Prop2v0 [F= (MAIN \ {| join,leave,lend,renew,return,reserve,take,cancel
       |})
```

Listing 31: Properties specification in fdr2 (part 1)

```
-- Property 3
-- An acquired book can be discarded only if it is neither lent nor reserved.
-- Check property in two parts
-- Part 0 - Check for loans

Prop3P0 = |||b:BOOKID @ Prop3P0Body(b)
Prop3P0Body(b) = acquire.b ->
      Prop3P0Loop(b); discard.b ->
      Prop3P0Body(b)
Prop3P0Loop(b) =
      [] m : MEMBERID @
         (
            (lend.m.b -> return.b -> Prop3P0Loop(b))
         []
            (take.m.b -> return.b -> Prop3P0Loop(b))
         )
   []
      SKIP

assert Prop3P0 [T= MAIN \ {| reserve,renew,cancel,join,leave |}

-- Part 1 - Check for reservations

Prop3P1 = |||b:BOOKID @ Prop3P1Body(b)
Prop3P1Body(b) = acquire.b ->
      (||| m : MEMBERID @ RESERVATION(m,b)); discard.b ->
```

```
    Prop3P1Body(b)

assert Prop3P1 [T= MAIN \ {| lend,renew,return,join,leave |}

-- Property 4. A person must be a member of the library in order to borrow a book.

Prop4 = |||m:MEMBERID @ Prop4Body(m)
Prop4Body(m) = join.m ->
    Prop4Loop(m); leave.m -
    > Prop4Body(m)
Prop4Loop(m) =
    ([] b : BOOKID @ (lend.m.b -> Prop4Loop(m)) [] (take.m.b -> Prop4Loop(m)))
  []
    SKIP

assert Prop4 [T= MAIN \ {| renew,return,reserve,cancel,acquire,discard |}
```

Listing 32: Properties specification in fdr2 (part 2)

```
-- Property 5. A book can be reserved only if it has been borrowed or already reserved by
-- another member.
-- Property 6. A book cannot be reserved by the member who is borrowing it.
-- Property 8. 8. A book cannot be lent to a member if it is reserved.
-- Property 9. A member cannot renew a loan if the book is reserved.
-- Property 10. A member is allowed to take a reserved book only if he owns the oldest reservation.
-- Property 11. A book can be taken only if it is not borrowed.
-- All these properties are checked using the same process expression, since
-- they require to express loand and reservation management

Prop5_6_8_9_10_11v0 = (|||b:BOOKID @ Prop5_6_8_9_10_11v0Body(b,<>,nullmid))
Prop5_6_8_9_10_11v0Body(b,l,cb) =
        ([] m:MEMBERID @ cb != m and
                        ( cb != nullmid
                        or
                          l != <>
                        ) and not elem(m,l)
                                        & reserve!m!b -> Prop5_6_8_9_10_11v0Body(b
                                ,l^<m>,cb)) -- property 5 & 6 [] ([]
    m:MEMBERID @ l != <> and m == head(l) and cb == nullmid
                                & take!m!b -> Prop5_6_8_9_10_11v0Body(b, tail(l),m)) --
                                    property 10 & 11
    [] ([] m:MEMBERID @ elem(m,l) & cancel!m!b -> Prop5_6_8_9_10_11v0Body(b, removex(l,m),cb))
    [] ( l == <> & lend?m!b -> Prop5_6_8_9_10_11v0Body(b,l,m)) -- property 8
    [] ( l == <> & renew!b -> Prop5_6_8_9_10_11v0Body(b,l,cb)) -- property 9
    [] ( cb != nullmid & return!b -> Prop5_6_8_9_10_11v0Body(b,l,nullmid))
```

```
assert Prop5_6_8_9_10_11v0 [T= MAIN \ {| acquire,discard,renew,join,leave |} -- Property 7. A book cannot be

reserved by a member who is reserving it.

Prop7v0 = (|||b:BOOKID @ |||m:MEMBERID @ Prop7v0Body(b,m))
Prop7v0Body(b,m) = reserve.m.b ->
      ( take.m.b -> Prop7v0Body(b,m)
      [] cancel.m.b -> Prop7v0Body(b,m)
      )

assert Prop7v0 [T= MAIN \ {| acquire,discard,lend,renew,return,join,leave |}
```

Listing 33: Properties specification in fdr2 (part 3)

```
-- Property 12. Anyone who has reserved a book can cancel the reservation at anytime before he takes it.

Prop12 = (|||b:BOOKID @ (|||m:MEMBERID @ Prop12Body(b,m)) |[ {| reserve,take, cancel |} ]|
       Prop12ResList(b,<>))
Prop12Body(b,m) = reserve.m.b ->
      ( take.m.b -> Prop12Body(b,m)
      [] cancel.m.b -> Prop12Body(b,m)
      )

Prop12ResList(b,l) =
        ([] m:MEMBERID @ not elem(m,l) & reserve!m!b -> Prop12ResList(b,l^<m>))
   [] ([] m:MEMBERID @ l != <> and m == head(l) & take!m!b -> Prop12ResList(b, tail(l)))
   [] ([] m:MEMBERID @ elem(m,l) & cancel!m!b -> Prop12ResList(b,removex(l,m)))

assert Prop12 [F= MAIN \ {| join,leave,acquire,discard,lend,renew,return |}

-- Property 13. A member can relinquish library membership only when all his loans have been
-- returned and all his reservations have either been used or canceled.

Prop13 = |||m:MEMBERID @ Prop13Body(m)
Prop13Body(m) = join.m ->
      (|||b:BOOKID @ Prop13Loop(m,b)); leave.m ->
      Prop13Body(m)
Prop13Loop(m,b) =
        (lend.m.b -> return.b -> Prop13Loop(m,b))
   []
        ( reserve.m.b ->
              (
                (take.m.b -> return.b -> Prop13Loop(m,b))
              []
                (cancel.m.b -> Prop13Loop(m,b))
              )
        )
   []
        SKIP
```

```
assert Prop13 [T= MAIN \ {| acquire, discard, renew |}
```

Listing 34: Properties specification in fdr2 (part 4)

```
-- Property 14. Ultimately, there is always a procedure that enables a member to leave the library.

Prop14 = (|||m:MEMBERID @ Prop14Body(m)) ||| CHAOS({| acquire, discard |})
Prop14Body(m) = join.m ->
      leave.m ->
      Prop14Body(m)

assert Prop14 [F= (MAIN [| RESERVSET |] STOP) \ BOOKMEMBERASSOC assert Prop14 [F= (MAIN [|
LOANSET |] STOP) \ BOOKMEMBERASSOC

-- Property 15. A member cannot borrow more than the loan limit defined at the system level for all users.

Prop15 = |||m:MEMBERID @ Prop15Body(m,{})

Prop15Body(m,s) =
          ([] b:BOOKID @ card(s) < maxNbLoans and not member(b,s) & lend!m!b -> Prop15Body(m,union(s,{b})) )
      [] ([] b:BOOKID @ card(s) < maxNbLoans and not member(b,s) & take!m!b ->
            Prop15Body(m,union(s,{b})) )
      [] ([] b:BOOKID @ member(b,s) & return!b -> Prop15Body(m,diff(s,{b})) )

assert Prop15 [T= MAIN \ {| join,leave,acquire,discard,reserve,cancel,renew |}
```

Listing 35: Properties specification in fdr2 (part 5)

```
-- Utility definitions

-- removes x from sequence l
removex(l,x) = if (l == <>)
         then <>
      else if (x == head(l))
         then tail(l) else
         <head(l)>^removex(tail(l),x)
```

Listing 36: Utility definitions for fdr2 specification

D cadp

```
specification bibli_main [ACQ, DISC, JOIN, LEAVE, LEND,
  RENEW, RET, RESERVE, CANCEL, TAKE] : exit behaviour
     (
     BOOK [ACQ, DISC, LEND, TAKE, RENEW, RET, RESERVE, CANCEL] (b1)
```

```
    |||
    BOOK [ACQ, DISC, LEND, TAKE, RENEW, RET, RESERVE, CANCEL] (b2)
    |||
    BOOK [ACQ, DISC, LEND, TAKE, RENEW, RET, RESERVE, CANCEL] (b3) )
  ||[LEND, TAKE, RENEW, RET, RESERVE, CANCEL]||
    (
    MEMBER [JOIN, LEAVE, LEND, TAKE, RENEW, RET, RESERVE, CANCEL] (m1)
    |||
    MEMBER [JOIN, LEAVE, LEND, TAKE, RENEW, RET, RESERVE, CANCEL] (m2)
    |||
    MEMBER [JOIN, LEAVE, LEND, TAKE, RENEW, RET, RESERVE, CANCEL] (m3)
    ) where library LNT_V1, BIBSEC endlib
endspec
```

Listing 37: The main behavior of the lotos-nt specification

```
module BIBSEC is

type BOOK_ID is b1, b2,
  b3 with "eq", "ne"
end type

type MEMBER_ID is m1, m2,
  m3, mnil with "eq", "ne"
end type

type MEMBER_LIST is
  NIL,
  CONS (HD : MEMBER_ID, TL : MEMBER_LIST) with "eq", "ne"
end type
```

Listing 38: The type definition of the lotos-nt specification

```
(* Removes all occurences of a member in the list *) function ml_remove(m: MEMBER_ID, l:
MEMBER_LIST) : MEMBER_LIST is
  case l in
    var temp_mem: MEMBER_ID, temp_list: MEMBER_LIST in NIL -> return NIL
    | CONS(temp_mem, temp_list) where (temp_mem eq m) -> return ml_remove(m,
      temp_list)
    | CONS(temp_mem, temp_list) -> return CONS(temp_mem,
      ml_remove(m, temp_list))
  end case
end function

(* Returns true if the member is the oldest in the reservation queue,
```

```
 * (returns False if the member is not in the queue) *) function ml_reserver_is_oldest(m: MEMBER_ID,
l: MEMBER_LIST) : BOOL is
   case l in
      var temp_mem: MEMBER_ID, temp_list: MEMBER_LIST in
      NIL -> return false
      | CONS(temp_mem, temp_list) where (temp_mem eq m) -> return true
      | ANY MEMBER_LIST -> return false end case
end function

(* Adds a member to the reservation queue *)
function ml_add_reserver(m: MEMBER_ID, l: MEMBER_LIST) : MEMBER_LIST is
   case l in
      var temp_mem: MEMBER_ID, temp_list: MEMBER_LIST in
      NIL -> return CONS(m, NIL)
      | CONS(temp_mem, temp_list) -> return CONS(temp_mem,
        ml_add_reserver(m, temp_list))
   end case
end function

(* Returns true if the list is empty *) function ml_is_nil(l:
MEMBER_LIST) : BOOL is
   case l in
      NIL -> return true
      | ANY MEMBER_LIST -> return false end case
end function
```

Listing 39: The function definition of the lotos-nt specification

```
process loan[LEND, TAKE, RENEW, RET:ANY]
(bid: BOOK_ID, mid: MEMBER_ID) is select
      LEND ( !bid, !mid )
   []
      TAKE ( !bid, !mid ) end
   select ;
   loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
      RENEW ( !bid, !mid ) end select end loop (* End
   Kleene *)
   ;
   RET ( !bid, !mid ) end process

process reservation[TAKE, RESERVE, CANCEL:ANY]
(bid: BOOK_ID, mid: MEMBER_ID) is RESERVE (
   !bid, !mid ) ; select
      TAKE ( !bid, !mid )
   []
      CANCEL ( !bid, !mid ) end select
```

```
end process
```

Listing 40: The behavior of the lotos-nt specification (loan and reservation)

```
process member[JOIN, LEAVE, LEND, TAKE, RENEW,
RET, RESERVE, CANCEL:ANY](mid: MEMBER_ID) is loop LOOP_1 in
  JOIN ( !mid ) ; par TAKE
      in par
          loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
            loan[LEND, TAKE, RENEW, RET](b1, mid)
          end select end loop (* End Kleene *)
          || loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
            loan[LEND, TAKE, RENEW, RET](b2, mid)
          end select end loop (* End Kleene *)
          || loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
            loan[LEND, TAKE, RENEW, RET](b3, mid)
          end select end loop (* End Kleene *) end par
      || par
          loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
            reservation[TAKE, RESERVE, CANCEL](b1, mid)
          end select end loop (* End Kleene *)
          || loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
            reservation[TAKE, RESERVE, CANCEL](b2, mid)
          end select end loop (* End Kleene *)
          || loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
            reservation[TAKE, RESERVE, CANCEL](b3, mid)
          end select end loop (* End Kleene *) end par end par
  ;
  LEAVE ( !mid ) end loop
end process
```

Listing 41: The behavior of the lotos-nt specification (member)

```
process book[ACQ, DISC, LEND, TAKE, RENEW,
RET, RESERVE, CANCEL:ANY](bid: BOOK_ID) is par ACQ, DISC, LEND, TAKE, RENEW,
  RET, RESERVE, CANCEL in
    bookNominal[ACQ, DISC, LEND, TAKE, RENEW, RET, RESERVE,
    CANCEL](bid)
  || bookController[ACQ, DISC, LEND, TAKE, RENEW, RET, RESERVE, CANCEL](bid)
  end par end process

process bookNominal[ACQ, DISC, LEND, TAKE, RENEW,
RET, RESERVE, CANCEL:ANY](bid: BOOK_ID) is loop LOOP_1 in
  ACQ ( !bid ) ; par TAKE in
        loop LOOP_2 in select break LOOP_2 [] (* Kleene *) var mid:MEMBER_ID in
            mid := any MEMBER_ID ; loan[LEND, TAKE, RENEW, RET](
            bid, mid )
```

```
            end var end select end loop (* End Kleene *)
        || par
            loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
              reservation[TAKE, RESERVE, CANCEL](bid, m1)
            end select end loop (* End Kleene *)
          || loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
              reservation[TAKE, RESERVE, CANCEL](bid, m2)
            end select end loop (* End Kleene *)
          || loop LOOP_2 in select break LOOP_2 [] (* Kleene *)
              reservation[TAKE, RESERVE, CANCEL](bid, m3)
            end select end loop (* End Kleene *) end par end par
  ;
  DISC ( !bid ) end loop
end process
```

Listing 42: The behavior of the lotos-nt specification (book, part 1)

```
process bookController[ACQ, DISC, LEND, TAKE, RENEW, RET, RESERVE,
CANCEL:ANY](bid: BOOK_ID) is var
res_queue: MEMBER_LIST, temp_member:
MEMBER_ID, current_borrower: MEMBER_ID in
  res_queue := NIL; current_borrower
  := mnil; loop LOOP_1 in select
    ACQ ( !bid )
    []
    DISC ( !bid )
    []
    LEND ( !bid, ?current_borrower) where (ml_is_nil(res_queue))
    []
    RENEW ( !bid, ?ANY MEMBER_ID ) where (ml_is_nil(res_queue))
    []
    RET ( !bid, ?ANY MEMBER_ID ) ; current_borrower := mnil
    []
    RESERVE ( !bid, ?temp_member ) where
      ( ( (current_borrower ne mnil) and
        (temp_member ne current_borrower) ) or
      ( (current_borrower eq mnil) and not
        (ml_is_nil(res_queue)) ) ) ;
      res_queue:= ml_add_reserver(temp_member, res_queue)
    []
    TAKE ( !bid, ?temp_member ) where
      ( (current_borrower eq mnil) and
      (ml_reserver_is_oldest(temp_member, res_queue) eq true) ); current_borrower :=
      temp_member ; res_queue := ml_remove(temp_member, res_queue)
    []
    CANCEL ( !bid, ?temp_member ) ; res_queue :=
    ml_remove(temp_member, res_queue) end select
```

```
end loop end var end process end module
```

Listing 43: The behavior of the lotos-nt specification (book, part 2)

```
(* Weak until operators *)

(*
  *   E [f_a W_b g]
  *   [[ E [f_a W_b g] ]] = { p | exists s in Path (p) such that
  *   ( exists k > 0 such that
  *   (forall i=0..(k-2), s(i)-l->s(i+1) => l in [[a]] or {tau}) and
  *   (forall i=0..(k-1), s(i) in [[f]]) and
  *   (s(k-1)-l->s(k) => l in [[b]]) and
  *   (s(k) in [[g]]) ) or
  *   ( forall k >=0,
  *   (s(k)-l->s(k+1) => l in [[a]] or {tau}) and (s(i) in [[f]]) ) }
  *)
macro EW_A_B (F, A, B, G) =
      EU_A_B ((F), (A), (B), (G)) or EG (Dia((A), true)) end_macro

(*
  *   A [f_a W_b g]
  *   [[ A [f_a W_b g] ]] = { p | forall s in Path (p),
  *   ( exists k > 0 such that
  *   (forall i=0..(k-2), s(i)-l->s(i+1) => l in [[a]] or {tau}) and
  *   (forall i=0..(k-1), s(i) in [[f]]) and
  *   (s(k-1)-l->s(k) => l in [[b]]) and
  *   (s(k) in [[g]]) ) or
  *   ( forall k >=0,
  *   (s(k)-l->s(k+1) => l in [[a]] or {tau}) and (s(i) in [[f]]) ) }
  *) def AW_A_B (F : stateset, A, B : labelset, G : stateset) : stateset = not(lfp(X,
      not(F) or Dia(not(A) and not(TAU) and not(B), true) or
      Dia(not(A) and not(TAU), not(G)) or
      Dia(not(B), X) or Dia(true, not(G) and X)
    )) end_def
```

Listing 44: Expression of the *weak until* operator in XTL with their denotational definition

```
(* Action A cannot be performed between action B and action C *) macro no_A_between_B_and_C(A,
B, C) =
   AG (Box((B), AW_A_B(true, not(A), (C), true))) end_macro

(* Action A cannot be performed outside action B and action C *) macro no_A_outside_B_to_C(A, B,
C) =
```

```
   ( INIT implies AW_A_B(true, not(A), (B), true) ) and
   AG (Box((C), AW_A_B(true, not(A), (B), true))) end_macro
```

Listing 45: Expression of the two temporal patterns in XTL

```
PRINT_FORM (
" 1. A book can be acquired by the library if it is not currently
       acquired --> ",
   INIT implies
                        EF(Dia(ACQ_B1, true)) and AG(Box(DISC_B1, EF(Dia(ACQ_B1, true)) ))
)

PRINT_FORM (
" 2. A book can only be acquired if it is not currently acquired
       --> ",
   no_A_between_B_and_C(ACQ_B1, ACQ_B1, DISC_B1)
)

PRINT_FORM (
" 3. An acquired book can be discarded, but only if it has not been lent or
       reserved --> ",
   INIT implies no_A_between_B_and_C(DISC_B1, LEND_B1_M1, RET_B1) and
   INIT implies no_A_between_B_and_C(DISC_B1, RESERVE_B1_M1,
      CANCEL_B1_M1 or TAKE_B1_M1)
)

PRINT_FORM (
" 4. An individual must be a member of the library in order to borrow a
       book. --> ",
   no_A_outside_B_to_C(LEND_B1_M1 or RESERVE_B1_M1, JOIN_M1, LEAVE_M1)
)

PRINT_FORM (
" 6. A book cannot be reserved by the member who is borrowing it. --> ", no_A_between_B_and_C(RESERVE_B1_M1, LEND_B1_M1
                                                                    or TAKE_B1_M1, RET_B1)
)

PRINT_FORM (
" 7. A book cannot be reserved by a member who is reserving it. --> ", no_A_between_B_and_C(RESERVE_B1_M1,
   RESERVE_B1_M1,
      TAKE_B1_M1 or CANCEL_B1_M1)
)
```

Listing 46: List of XTL properties, part 1

```
PRINT_FORM (
" 8. A book cannot be lent to a member if it is reserved. --> ", no_A_between_B_and_C(
   LEND_B1_M1, RESERVE_B1_M1, TAKE_B1_M1 or CANCEL_B1_M1 )
)

PRINT_FORM (
```

```
" 9. A member cannot renew a loan if the book is reserved. --> ", no_A_between_B_and_C(RENEW_B1, RESERVE_B1,
   TAKE_B1 or CANCEL_B1)
)

PRINT_FORM (
" 11. A book can be taken only after it is returned. --> ", no_A_between_B_and_C(TAKE_B1, TAKE("B1", "M2"),
   RET_B1)
)

PRINT_FORM (
" 12. Anyone who has reserved a book can cancel the reservation at anytime before the
        reservation has been used, i.e. before the member takes it. --> ",
   AG( Box(RESERVE_B1_M1,
        AW_A_B( Dia(CANCEL_B1_M1, true), true,
                CANCEL_B1_M1 or TAKE_B1_M1, true )))
)

PRINT_FORM (
" 13. A member can relinquish library membership only when all his loans have been returned and all
        his reservations have either been used or canceled. --> ",
   no_A_between_B_and_C( LEAVE_M1, LEND_B1_M1 or TAKE_B1_M1, RET_B1 ) and
   no_A_between_B_and_C( LEAVE_M1, RESERVE_B1_M1, CANCEL_B1_M1 or TAKE_B1_M1 )
)

PRINT_FORM (
" 14. Ultimately, a member can leave the library. --> ", AG( Box(JOIN_M1,
        AW_A_B(EF(Dia(LEAVE_M1, true)), true, LEAVE_M1, true)))
)

PRINT_FORM (
" 15. A member cannot borrow more than the loan limit (two books). -->", not( EF(Dia(BORROW_M1,
   EF_A(not(RET_M1), Dia(BORROW_M1, EF_A(not(RET_M1), Dia(BORROW_M1, true)) )))))
)
```

Listing 47: List of XTL properties, part 2

E Alloy

```
module model open util/sequniv as
ResSq

------------------------------
-------Declaration----------------------------------------
one sig Constants
{ maxNbLoans : Int
}
{ maxNbLoans = 7 } sig
Book{} sig Member{}
```

```
 sig Lib
{ members:set Member, books: set Book , loan: (books -> members),
   membersReservingOneBook: (seq members)->books, Renew:
   (books -> members)
}

/* ================================
      = List of no change predicates =
      = They are used in action to describe which state =
      = variables remain unchanged = ================================*/

pred NoChangebooks[L,L':Lib]
{
   L.books =L'.books }

pred NoChangemembers[L,L':Lib]
{
   L.members =L'.members }

pred NoChangeloan[L,L':Lib]
{
   L.loan=L'.loan
}
```

Listing 48: Library specification in Alloy (part 1)

```
pred NoChangeSeqBook[L,L':Lib]
{
   L.membersReservingOneBook= L'.membersReservingOneBook }


pred NochangeRenew[L,L':Lib]
{
   L.Renew = L'.Renew
}

/*----------------
      Initialisation ----------*/
pred Init [L:Lib]
{ no L.books no L.members no L.loan no
   L.membersReservingOneBook no
   L.Renew
}

/*--------- ---------Acquire ----------*/
pred CanBeAcquire[L:Lib,b:Book]
{ no(b & L.books) // verify that b is not in the Library
}
```

```
pred Acquire[b:Book,L,L':Lib]
{
   ----Preconditon-------
   CanBeAcquire[L,b]

   -----Postcondition-------
   L'.books = L.books + b // add the b in the set of books

   ----NoChanges-----
   NoChangemembers[L,L']
   NoChangeloan[L,L']
   NoChangeSeqBook[L,L']
   NochangeRenew[L,L']
}
```

Listing 49: Library specification in Alloy (part 2)

```
/*--------- ----------
      Join
----------*/ pred
CanJoin[m:Member,L:Lib]
{ no (m & L.members)// m does not exist in the Library.
}

pred Join[m:Member,L,L':Lib]{

   ----Precondition-----
   CanJoin[m,L]

   -----Postcondition------
   L'.members=L.members +m// add the m in the set of members

   ------Nochanges-----
   NoChangebooks[L,L']
   NoChangeloan[L,L']
   NoChangeSeqBook[L,L']
   NochangeRenew[L,L']
}

/*--------- ----------
      LEND
----------*/ pred
CanLend[m:Member,b:Book,L:Lib]
{
   (b in L.books) and (m in L.members) // b and m are in the Library
   (#((L.loan).m)<Constants.maxNbLoans) //maxNbLoans is the number maximum of loans authorized
   all m':Member|no((L.loan).m' & b)// b is not lent (no
   (L.membersReservingOneBook.b))// b not reserved
}
```

```
pred Lend[m:Member,b:Book,L,L':Lib]
{ -----Precondition------------
   CanLend[m,b,L]

   ----Postcondition------------L'.loan=L.loan + (b->m)

   ----Nochanges------------
   NoChangebooks[L,L']
   NoChangemembers[L,L']
   NoChangeSeqBook[L,L']
   NochangeRenew[L,L']
}
```

Listing 50: Library specification in Alloy (part 3)

```
/*--------- ----------
      RESERVE
-------------------*/ pred
CanReserve[m:Member,b:Book,L:Lib]
{
   (b in L.books and m in L.members ) // b and m are in the Library one (b & ((L.loan).Member)) or (some
   (L.membersReservingOneBook.b))// the book is a borrowed
      no (m & b.(L.loan)) // m is not lent
   no (Int.(L.membersReservingOneBook.b) & m) //it can't be reserved more than one Time by the same member
}


pred Reserve[m:Member,b:Book,L,L':Lib]
{
   ---- Precondition----
   CanReserve[m,b,L]

   ------PostCondition-----
   L'.membersReservingOneBook.b = ResSq/add[L.membersReservingOneBook.b,m]

   -----Nochanges-------
   all b':Book - b|L'.membersReservingOneBook.b' = L.membersReservingOneBook.b'
   NoChangebooks[L,L']
   NoChangemembers[L,L']
   NoChangeloan[L,L']
   NochangeRenew[L,L']
}
```

Listing 51: Library specification in Alloy (part 4)

```
/*--------- ----------
      CANCEL
-------------------*/ pred
CanCancel[m:Member,b:Book,L:Lib]
{
```

```
   (b in L.books and m in L.members ) // // b and m are in the Library one (Int->m &
    (L.membersReservingOneBook.b))// b is reserved by m
}

pred Cancel[m:Member,b:Book,L,L':Lib]
{ --------Preconditon----------
   CanCancel[m,b,L]

    --------Postconditon------------
   L'.membersReservingOneBook.b=ResSq/delete[L.membersReservingOneBook.b,ResSq/ indsOf
         [L.membersReservingOneBook.b,m]]// delete m from the list of reservation of b

   ------Nochanges--------
   all b':Book - b|L'.membersReservingOneBook.b' = L.membersReservingOneBook.b'
   NoChangebooks[L,L']
   NoChangemembers[L,L']
   NoChangeloan[L,L']
   NochangeRenew[L,L']

}
```

Listing 52: Library specification in Alloy (part 5)

```
/*--------- ----------
     RETURN ----------*/

pred CanReturn[m:Member,b:Book,L:Lib]
{
   (b in L.books and m in L.members ) one ((L.loan).m & b) // b is
   already lent to m
}


pred Return[m:Member,b:Book,L,L':Lib]
{
   ----Precondition-----
   CanReturn[m,b,L]

   ----PostConditon--------
   L'.loan=L.loan - (b ->m) // delete the b->m from the set of loans
   L'.Renew = L.Renew - (b -> m)// same thing

   ----Nochanges--------
   NoChangebooks[L,L']
   NoChangemembers[L,L']
   NoChangeSeqBook[L,L']


}
```

Listing 53: Library specification in Alloy (part 6)

```
/*--------------------
   TAKE ----------------------*/ pred
CanTake[m:Member,b:Book,L:Lib]
{
   (b in Lib.books) and (m in L.members)// b and m are in the Library
   (#((L.loan).m)<Constants.maxNbLoans) //maxNbLoans is the number maximum of lend authorized
   (L.membersReservingOneBook.b) = (0 -> m) // m is first in the list of reservation
   no (b.(L.loan)) // the book is not lent
}

pred Take[m:Member,b:Book,L,L':Lib]
{
   -----Preconditon-------
   CanTake[m,b,L]

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

   -----Nochanges-------
   all b':Book - b|L'.membersReservingOneBook.b' = L.membersReservingOneBook.b'
   NoChangebooks[L,L']
   NoChangemembers[L,L']
   NochangeRenew[L,L']
}
```

Listing 54: Library specification in Alloy (part 7)

```
/*-----------------
      LEAVE
------------------*/

pred CanLeave[m:Member,L:Lib]
{ m in L.members no (L.loan.m) // m is not in the lent list no( Int.(L.membersReservingOneBook.Book) &
   m)// m has no reseravation
}

pred Leave[m:Member,L,L':Lib]
{
   ------Preconditon-------
   CanLeave[m,L]

   ------Postconditon-----L'.members = L.members -
   m

   ----Nochanges---------
   NoChangeloan[L,L']
   NochangeRenew[L,L']
   NoChangeSeqBook[L,L']
```

```
      NoChangebooks[L,L']
}

/*-----------------
      DISCARD --------------------*/

pred CanDiscard[b:Book,L:Lib]
{ b in L.books no (b.(L.loan)) no
   ((L.membersReservingOneBook.b) )
}

pred Discard[b:Book,L,L':Lib]
{
   ------Precondition-------
   CanDiscard[b,L]
   ------Postconditon--------
   L'.books = L.books - b
   -----Nochanges-------
   NoChangeloan[L,L']
   NoChangeSeqBook[L,L']
      NoChangemembers[L,L']
   NochangeRenew[L,L']
}
```

Listing 55: Library specification in Alloy (part 8)

```
/*--------------------
      RENEW ----------------------*/ pred
CanRenew[m:Member,b:Book,L:Lib]
{ one (b.(L.loan) & m) // b is already borrowed by m
   ResSq/isEmpty [L.membersReservingOneBook.b] //b has no reservation
}

pred Renew[m:Member,b:Book,L,L':Lib]
{
   ------Preconditon-------
   CanRenew[m,b,L]

   -----Postcondition--------
   L'.Renew=L.Renew ++ (b->m) // override the old b->m

   ------Nochanges-----
   NoChangebooks[L,L']
   NoChangemembers[L,L']
   NoChangeloan[L,L']
   NoChangeSeqBook[L,L']
}
```

Listing 56: Library specification in Alloy (part 9)

```
/*
Here we can find all the proprieties that we have arrived to expresse Expect
14 and 15 which are in the second File
*/

open LibSpecification

--1. A book can always be acquired by the library when it is not currently acquired.
--2. A book cannot be acquired by the library if it is already acquired.
assert Prop1And2
{

   all b:Book,L:Lib
   {
      CanBeAcquire[L,b] <=> not (b in L.books) // 1 and 2 }
} check Prop1And2 for 2 Lib,8 Member,8 Book
//It has been impossible to express the prop1 in this form!!!!!!

/*assert Prop1And2
{

   all b:Book,L:Lib
   {

      some L':Lib| Acquire[b,L,L'] => not (b in L.books) not (b in L.books) =>some L':Lib| Acquire[b,L,L'] Can not
   expressed because we have to generate the transition graph }
} check Prop1And2 for 2 Lib,8 Member,8 Book
*/
```

Listing 57: Properties specification in Alloy (part 1)

```
--An acquired book can be discarded only if it is neither lent nor reserved.
assert Prop3
{ all b:Book,L:Lib
   { some L':Lib |Discard[b,L,L'] => no (b.(L.loan)) and no ((L.
               membersReservingOneBook.b) )
   }
} check Prop3 for 2 Lib,8 Member,8 Book

--A person must be a member of the library in order to borrow(lend or take) a book. assert Prop4 { all
b:Book,m:Member,L:Lib
   { some L':Lib|Lend[m,b,L,L'] => (b in L.books) and (m in L.members)// b and m in the Library
      some L':Lib|Take[m,b,L,L'] => (b in L.books) and (m in L.members) }
} check Prop4 for 2 Lib,8 Member,8 Book

--A book can be reserved only if it has been borrowed or already reserved by another member
assert Prop5
{ all b:Book,m:Member,L:Lib
   { some L':Lib|Reserve[m,b,L,L'] => (b in ((L.loan).Member)) or (some (L.
            membersReservingOneBook.b))
```

```
   }
}
check Prop5 for 2 Lib,8 Member,8 Book

--A book cannot be reserved by the member who is borrowing it.
assert Prop6
{ all b:Book,m:Member,L:Lib
   { some L':Lib|Reserve[m,b,L,L'] => ((m !in b.(L.loan))) }
}
check Prop6 for 2 Lib,8 Member,8 Book
```

Listing 58: Properties specification in Alloy (part 2)

```
--A book cannot be reserved by a member who is reserving it.
assert Prop7
{ all b:Book,m:Member,L:Lib
   { some L':Lib|Reserve[m,b,L,L'] => ( (m !in Int.(L.membersReservingOneBook.b))
            )
   }
} check Prop7 for 2 Lib,8 Member,8 Book

--A book cannot be lent to a member if it is reserved assert Prop8
{ all b:Book,m:Member,L:Lib
   { some L':Lib|Lend[m,b,L,L'] => (no (L.membersReservingOneBook.b)) }
}

check Prop8 for 2 Lib,8 Member,8 Book

--A member cannot renew a loan if the book is reserved assert Prop9
{ all b:Book,m:Member,L:Lib
   { some L':Lib|Renew[m,b,L,L'] => (no (L.membersReservingOneBook.b)) }
}

check Prop9 for 2 Lib,8 Member,8 Book

--A member is allowed to take a reserved book only if he owns the oldest reservation
assert Prop10
{ all b:Book,m:Member,L:Lib
   { some L':Lib|Take[m,b,L,L'] => (L.membersReservingOneBook.b) = (0 -> m) }
}
check Prop10 for 2 Lib,8 Member,8 Book
```

Listing 59: Properties specification in Alloy (part 3)

```
--A book can be taken only if it is not borrowed assert Prop11
{ all b:Book,m:Member,L:Lib
   { some L':Lib|Take[m,b,L,L'] => not ( b in (L.loan).Member) }
} check Prop11 for 2 Lib,8 Member,8 Book

--Anyone who has reserved a book can cancel the reservation at anytime before he takes it
assert Prop12
```

```
{ all b:Book,L:Lib,m:Member
   { some L':Lib| Take[m,b,L,L'] => CanCancel[m,b,L'] }
} check Prop12 for 2 Lib,8 Member,8 Book

--A member can relinquish library membership only when all his loans have been returned and all his reservations
       have either been used or canceled
assert Prop13
{ all m:Member,L:Lib
   { some L':Lib|Leave[m,L,L'] => (no (L.loan.m) and (m !in Int.(L.
            membersReservingOneBook.Book)))
   }
} check Prop13 for 2 Lib,8 Member,8 Book

--A member cannot borrow more than the loan limit defined at the system level for all users
assert Prop15
{ all L,L':Lib,m:Member,b:Book
   {
      Take[m,b,L,L'] => #(L'.loan.m)<=7
      Lend[m,b,L,L'] => #(L'.loan.m)<=7
   }
}

check Prop15 for 2 Lib, 8 Member, 8 Book
```

Listing 60: Properties specification in Alloy (part 4)

```
open LibSpecification open util/IJALSequence[Lib] as
LibSeq1 open util/LCRSequence[Lib] as LibSeq2

-------------------------------------------------------
----------------------TRACES--------------------------
-------------------------------------------------------

fact {

  Init[LibSeq1/first[Seq]]

   all idx : LibSeq1/inds[Seq] - LibSeq1/lastIdx[Seq]| some m:Member,b:Book|
         Join[m,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]] or
         Acquire[b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]] or
         Lend[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]] or
         Reserve[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]] or
         Take[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]] or
         Renew[m,b,at[Seq,idx],at[Seq,idx.LibSeq1/ord/next]]
}

-------------------------------------------------------
------------------Prop14(CTL)--------------------------------------------------------------------------------------
```

```
pred Buggy_leave[L,L':Lib]{L=L'}// Just For Test ------L=Leave,C=Cancel,R=Return---------

pred LCR[m:Member,L,L' : Lib]
{ some b:Book |
      Cancel[m,b,L,L'] or
      Return[m,b,L,L'] or
      Leave[m,L,L']
   //or Buggy_leave[L,L'] //for test

}

-----CanLCR=forces the analyser to go to the end of operation --pred CanLCR[L:Lib,m:Member]
{
   CanLeave[m,L] or some
   b:Book|CanCancel[m,b,L] or some b
   :Book|CanReturn[m,b,L]
}
```

Listing 61: Property 14 in Alloy (part 1)

```
-------TransLCR = is The action that the analyser is able to do in The SeqLCR (
       LCR)

pred TransLCR[m:Member,P:SeqLCR]
{ all idx : LibSeq2/inds[P]-LibSeq2/lastIdx[P] | // all Sequence's index except the last
                                                 LCR[m,LibSeq2/at[P,idx],LibSeq2/at[P,idx.LibSeq2/ord/next]]// LCR[L,L.next]
}


pred NegProp14 []
{

   one SeqLCR

    some P : SeqLCR|

   some m : LibSeq1/last[Seq].members |

            //LibSeq1/last[Seq] = LibSeq2/first[P] //the last elem in the Seq of IJAL is the first in the Seq of LCR
   //and
            TransLCR[m,P] //LCR and m in
   LibSeq2/last[P].members

   and
            (not CanLCR[LibSeq2/last[P],m] // in the last of SeqLCR we can not do
                  LCR or
            LibSeq2/lastIdx[P] = LibSeq2/finalIdx//the end of SeqLCR is the last Idx (max Seq)
            ) }
```

```
run{ NegProp14[]}for 3 Member,3 Book,36 Lib, 13 SeqIdx,13 SeqIdx1,13 SeqLCR
```

Listing 62: Property 14 in Alloy (part 2)

```
--The last Library in the IAJLR Sequence is th first Library in the LCR fact{
   LibSeq1/last[Seq] = LibSeq2/first[SeqLCR]
}
-------------------------------------------------------------------------
--Testing that in some cases running a model with a predicate ----------is the same things than checking an assert
and also testing the difference -----
--between their solving time.
-------------------------------------------------------------------------assert Prop14
{

all P : SeqLCR|
all m: LibSeq1/last[Seq].members|

      TransLCR[m,P]//OrLCR
=>

(
   (not m in LibSeq2/last[P].members)

or
   (
                                                ( CanLCR[LibSeq2/last[P],m] // incomplete sequence : still possible to do a
                  LCR and
         (not LibSeq2/lastIdx[P] = LibSeq2/finalIdx)//the sequence is not of maximal length

         )
   )
)

}

check Prop14 for 4 Member,4 Book, 50 Lib, 25 SeqIdx,25 SeqIdx1,25 SeqLCR
```

Listing 63: Property 14 in Alloy (part 3)

```
open LibSpecification open util/sequence[Lib] as
LibSeq
/*
Specification of property 14 using traces. A fact is used to define the states from which
the property must be satisfied (ie, the trace does not start from the initial state of the
library.

*/


pred BuggyLeave[L,L':Lib]{L=L'}//Just For Test
```

```
pred LCR[m:Member,L,L' : Lib]
{ some b:Book |
      Cancel[m,b,L,L'] or
      Return[m,b,L,L']
  //For test switch Leave and BuggyLeave or Leave[m,L,L']
  //or BuggyLeave[L,L']
}

pred TransLCR[m:Member,P:Seq]
{ all idx : LibSeq/inds[P]-LibSeq/lastIdx[P] |
     LCR[m,LibSeq/at[P,idx],LibSeq/at[P,idx.LibSeq/ord/next]] }

pred OrCanLCR[L:Lib,m:Member]
{ CanLeave[m,L] or some
  b:Book|CanCancel[m,b,L] or some b
  :Book|CanReturn[m,b,L]
}
```

Listing 64: Property 14 in Alloy, second version (part 1)

```
---------- Property 14 as a run ----------pred NegProp14 {

all P : Seq | Prop14StartLib[LibSeq/first[P]] // start from a valid library state

  some P : Seq |
  some m : LibSeq/first[P].members |

                    TransLCR[m,P] and m in
              LibSeq/last[P].members
              and
                ( not OrCanLCR[LibSeq/last[P],m] // sequence max : LCR can't be applyed
        or
              LibSeq/lastIdx[P] = LibSeq/finalIdx // Maximum sequence's length was reached
                )
}

// Property 14 is checked by executing a run using a fact which negates property 14
// m members, b books => at most l loans and b-l membersReservingOneBook
// hence, need l returns + (b-l) cancels + 1 leave =>
// b+1 actions => // trace of length b+2 run NegProp14 for
10 but 8 Book, 8 Member run NegProp14 for 8 but 6 Book, 6
Member run BuggyLeave for 5 run {} for 3 but 2 Lib
```

Listing 65: Property 14 in Alloy, second version (part 2)

```
-------------------- Property 14 using a check -------------------------
/*
This states property 14 as an assert and uses a check to verify it.
*/ assert Prop14 {

  all P : Seq |
```

```
  all m : LibSeq/first[P].members |
    (
      Prop14StartLib[LibSeq/first[P]] and
      TransLCR[m,P]
    )
  =>
  (
      (not m in LibSeq/last[P].members) or
      ( // case when the sequence is incomplete
          OrCanLCR[LibSeq/last[P],m] and
          not LibSeq/lastIdx[P] = LibSeq/finalIdx
      )
  )
}

//check {not NegProp14}for 8 but 6 Book, 6 Member //check Prop14 for
8 but 6 Book, 6 Member check Prop14 for 10 but 8 Book, 8 Member
```

Listing 66: Property 14 in Alloy, second version (part 3)

```
----------Define what are the valid first library state of a trace
      ----------pred
Prop14StartLib[L:Lib]
{

  all m: Member| (#((L.loan).m)<=Constants.maxNbLoans)

-- The book can not be reserved if it's not loaned or reserved all b : Book|
      some (L.membersReservingOneBook.b)
      =>
      ((b in L.loan.Member) or ((#L.membersReservingOneBook.b) > 0))

-- the Member can not reserve and lend the same book all m : Member,b : Book|
    (m in b.(L.loan) => not (m in Int.(L.membersReservingOneBook.b)))

--the same thing all m : Member,b :
  Book|
      (m in Int.(L.membersReservingOneBook.b)) => not (m in b.(L.loan))

-- the member can not Renew a book and he has not lend this book all b : Book,m : Member|
      (b in L.Renew.m) =>b in (L.loan.m)//7

-- books -> lone members all b : Book|
      ((b in L.books) and (b in L.loan.Member)) => one (b ->Member & (L.loan)) all b : Book|
      ((b in L.books) and (b in L.Renew.Member)) => one (b ->Member & (L.Renew))

//it defines the valid sequences of reservations for a book. all m:Member,b:Book |
    (m in Int.(Lib.membersReservingOneBook.b))
```

```
  => one (Int->m & (Lib.membersReservingOneBook.b))

}
```

Listing 67: Property 14 in Alloy, second version (part 4)

F ProB

```
MACHINE Library_LTL_Sym
    /*
    Specification of a library system
    University of Sherbrooke
    First version
    Eric Gaudet: July 21, 2009
    Modified by
    Marc Frappier and Benoit Fraikin: May 06, 2010
    Benoit Fraikin: May 28, 2010
    Michael Leuschel: June 6 2010 --> to be able to make Symmetry Reduction */
   /* for scope 3 the reduction is not dramatic, but still noticeable
      (391 seconds down to 221 seconds, 35544 nodes down to 18410 nodes,
      372716 down to 192938 transitions with hash symmetry) LTL assertion checking is
      then very quick */

SETS
    MEMBERID /* = {m1,m2,m3} made deferred for symmetry */ ;
    BOOKID /* = {b1,b2,b3} made deferred for symmetry */

CONCRETE_CONSTANTS
    /* Maximum of books a member can borrow */ maxNbLoans ,
    /* added constants used in LTL formulas */ m1, m2, b1

DEFINITIONS scope_MEMBERID == 3; scope_BOOKID == 3;
    BOOK_IS_NOT_RESERVED == reservation(book_) = [] ;
    BOOK_IS_RESERVED == reservation(book_) /= [] ;
    MEMBER_HAS_NO_RESERVATION == union(ran(reservation)) |> {member_} = {} ;
    Indice (b1,m1) == (reservation(b1)~)(m1) ;
    /* 1 - A book can be acquired by the library if it is not currently acquired
        . */
    /*ASSERT_LTL_1 == "G( {b1 /: book} => e(Acquire(b1)) )" ; */
    /* 2 - A book can be acquired by the library only if it is not currently acquired. */
    ASSERT_LTL_2 == "G( e(Acquire(b1)) => {b1 /: book})" ;
    /* 3 - An acquired book can be discarded, but only if it has not been lent or reserved. */
    ASSERT_LTL_3 == "G( {b1 : book} => ( e(Discard(b1)) => ({b1 /: dom(loan)} &
        {reservation(b1) = []}) ) )" ;
```

```
/* 4 - A person must be a member of the library in order to borrow a book.
     */
ASSERT_LTL_4 == "G( e(Lend(m1,b1)) => {m1 : member} )" ;
```

Listing 68: The B machine — headers — 1

```
/* 5 - A book can be reserved only if it has been lent or already reserved by another member. */
ASSERT_LTL_5 == "G( e(Reserve(m1,b1)) => ( {m1 /: ran(reservation(b1))} & { b1|->m1 /: loan} & {b1:dom(loan)}
     or {reservation(b1) = []} ) )" ;
/* 6 - A book cannot be reserved by the member who is borrowing it. */
ASSERT_LTL_6 == "G( e(Reserve(m1,b1)) => not({b1 |-> m1 : loan}) )" ;
/* 7 - A book cannot be reserved by the member who is reserving it. */
ASSERT_LTL_7 == "G( e(Reserve(m1,b1)) => not({m1 : ran(reservation(b1))}) )" ;
/* 8 - A book cannot be lent to a member if it is reserved. */
ASSERT_LTL_8 == "G( {b1 : dom(reservation)} => not(e(Lend(m1,b1))) )" ; /* 9 - A member cannot renew a
loan if the book is reserved. */
ASSERT_LTL_9 == "G( {b1 : dom(reservation)} => not(e(Renew(m1,b1))) )" ;
/* 10 - A member is allowed to take it only if he is the one who made the oldest reservation. */
ASSERT_LTL_10 == "G( e(Take(m1,b1)) => {reservation(b1) /= [] & first( reservation(b1)) = m1} )" ;
/* 11 - A book can be taken only after it is returned */
ASSERT_LTL_11 == "G( {b1 |-> m2 : loan} => not(e(Take(m1,b1))) W {b1 |-> m2 /: loan} )" ;
/* 12 - Anyone who has reserved a book can cancel the reservation at anytime before the reservation has been
      used, i.e. when the member takes it. */
/*ASSERT_LTL_12 == "G( {m1 : ran(reservation(b1))} => ([Take(m1,b1)] R e(
     Cancel(m1,b1))) )" ;*/ /* verifier qu'il ne faut pas un release */
/* 13 - A member can relinquish library membership only when all his loans have been returned and all his
     reservations have either been used or canceled */
ASSERT_LTL_13 == "G( e(Leave(m1)) => {m1 : member} & {m1 /: ran(loan)} & { union(ran(reservation)) |> {m1} =
     {}} )" ;
/* 14 - Ultimately, a member can leave the library. */
/* ASSERT_LTL_14 == "{1=1}" ; */ /* cannot be done in LTL -> reset property
     */
/* CTL version "AG ( EF ( {m1 /: member} ) )"
/* 15 - A member cannot borrow more than the loan limit defined at the system level for all users. */
ASSERT_LTL_15 == "G( {!xm.( xm:member => card(loan |> {xm}) <= maxNbLoans )}
      )" /* INVARIANT */
```

Listing 69: The B machine — headers — 2

```
PROPERTIES maxNbLoans = scope_BOOKID - 1 &
    m1 : MEMBERID & m2 : MEMBERID & m1 /=
    m2 & b1 : BOOKID

ABSTRACT_VARIABLES
    loan , member , book
    , reservation
```

```
INVARIANT member <: MEMBERID &
    book <: BOOKID & loan : book +->
    member &
    /*A member can't borrow more than the maxNbLoans of the system*/ ! mm . ( mm : member
    => card ( loan |> { mm } ) <= maxNbLoans ) & reservation : book --> iseq(member)

INITIALISATION
    loan := {} || book := {} ||
    member := {} ||
    reservation := {}
```

Listing 70: The B machine — headers — 3

```
    Acquire ( book_ ) =
    PRE book_ : BOOKID &
        book_ /: book
    THEN
        book := book \/ { book_ } ||
    reservation(book_) := [] END ;

    Cancel ( member_ , book_ ) =
    PRE member_ : MEMBERID & member_ : member &
        book_ : BOOKID & book_ : book & book_ :
        dom(reservation) & member_ :
        ran(reservation(book_))
    THEN reservation(book_) :=
                                    (reservation(book_) /|\ (Indice(book_, member_) - 1))
            ^
            (reservation (book_) \|/ Indice(book_, member_)) END ;

    Discard ( book_ ) =
    PRE book_ : BOOKID & book_ : book
        & book_ /: dom ( loan ) &
        BOOK_IS_NOT_RESERVED THEN
        book := book - { book_ } ||
        reservation := {book_} <<| reservation END ;

    Join ( member_ ) =
    PRE member_ : MEMBERID &
        member_ /: member
    THEN member := member \/ { member_ } END ;
```

Listing 71: The B machine — Operations — 1

```
    Leave ( member_ ) =
    PRE member_ : MEMBERID &
        member_ : member &
        member_ /: ran(loan) &
        MEMBER_HAS_NO_RESERVATION THEN
    member := member - { member_ } END ;
```

```
Lend ( member_ , book_ ) =
PRE member_ : MEMBERID &
    member_ : member & book_ :
    BOOKID & book_ : book & book_
    /: dom ( loan ) &
    BOOK_IS_NOT_RESERVED & card ( loan |> { member_ } ) <
    maxNbLoans
THEN loan ( book_ ) := member_ END ;

Renew ( member_ , book_ ) =
PRE member_ : MEMBERID & book_ : BOOKID &
    member_ : member & book_ : book & (
    book_ |-> member_ ) : loan &
    BOOK_IS_NOT_RESERVED
THEN skip
END ;
```

Listing 72: The B machine — Operations — 2

```
Reserve ( member_ , book_ ) =
PRE member_ : MEMBERID & book_ : BOOKID &
    member_ : member & book_ : book & member_ /:
    ran(reservation(book_)) & book_ |-> member_ /:
    loan &
    (
        book_ : dom ( loan )
        or BOOK_IS_RESERVED
    )
THEN reservation := reservation <+ { book_ |-> ((reservation(book_) <- member_ )) } END ;

Return ( book_ ) =
PRE book_ : BOOKID & book_ :
    book & book_ : dom ( loan )
THEN
    loan := { book_ } <<| loan END ;

Take ( member_ , book_ ) =
PRE member_ : MEMBERID & book_ : BOOKID & member_ :
    member & book_ : book & book_ /: dom ( loan ) & card ( loan
    |> { member_ } ) < maxNbLoans & size(reservation(book_)) /=
    0 & first(reservation(book_)) = member_
THEN
    loan ( book_ ) := member_ ||
    reservation := reservation <+ { book_ |-> tail(reservation(book_)) } END
```

Listing 73: The B machine — Operations — 3