Comparison of Model Checking Tools for Information Systems

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

GRIL, Université de Sherbrooke, Québec, Canada
benoit.fraikin@usherbrooke.ca

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.

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.

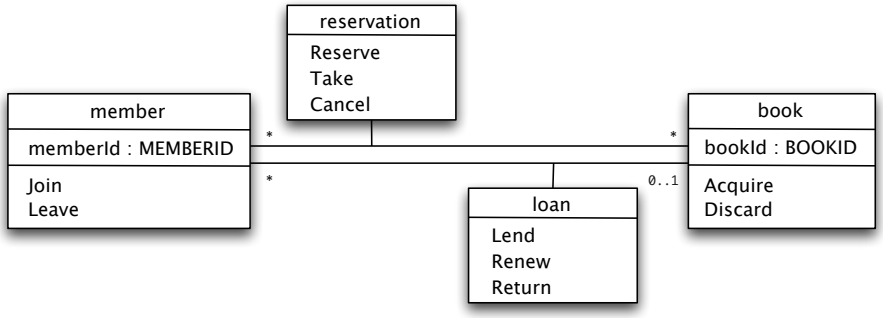

Fig. 1. Requirement class diagram of the library system

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 on-the-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 an 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 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 a 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 refer to state variables; LTL properties can refer to both state variables and input variables. 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 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: $\sqsubseteq_T$ (trace refinement), $\sqsubseteq_F$ (stable-failure refinement) and $\sqsubseteq_{FD}$ (stable-failure-divergence refinement). We say that $P \sqsubseteq_T Q$ 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 $P \sqsubseteq_F Q$ 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 up 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 actions B and C. Therefore the two main patterns are $\Box(\texttt{can\_}A \Rightarrow P)$ or $\Box(P \Rightarrow \texttt{can\_}A)$. 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 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: $Nominal = E_1 \parallel \ldots \parallel E_n$. Since CSP_M does not directly implement Hoare's parallel operator $\parallel$, we use CSP_M's operator $\overset{\parallel}{x}$, 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 $\boldsymbol{v}$ of an entity E are represented by a recursive process $C_E(k, \boldsymbol{v})$ which offers a choice [ ] between actions to control, in the form $[\,]_i \ G \ \& \ a_i \rightarrow C_E(k, e(\boldsymbol{v}))$, where "&" denotes a guard operator with condition G which tests the values of $\boldsymbol{v}$. 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: $P \sqsubseteq_T \ Q \setminus (\Sigma - \{a_i\})$, where $\setminus$ 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: acquire(b)→discard(b)→ $P(b)$. 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 $\mathtt{e}_i$ models the active instances of E_i in a state, and $\mathtt{a}_i$ models the instances of association A_i. Each action is represented by a predicate $P[s : S, s' : S, p : T]$ relating a before-state s to an after-state s' 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 $\forall s, s', p : T : S \cdot b[s, s', p] \Rightarrow preA[s', p]$. 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 $\exists s' : S \cdot postA[s, s', p]$ 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, unless the system model is small enough to fit within memory bounds; if the bounds provided for the check command do not include the full state space, then Alloy can always find a small model where the desired state s' is not included. 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 $\forall s, s' : S, p : T \cdot I[s] \wedge a[s, s', p] \Rightarrow I[s']$. 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: $\Box(\texttt{can\_}A \Rightarrow P)$ and $\Box(P \Rightarrow \texttt{can\_}A)$. 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 can handle up to 98 instances for all properties except 14 in less than a minute, because it only needs to cover a small subset of the state space to check these properties. 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

Fig. 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, Universit de Sherbrooke (2010), available 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 vrification de programmes LOTOS. Ph.D. thesis, Universit 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)