Under consideration for publication in Formal Aspects of Computing

Verification of EB^3 Specifications using CADP

Dimitris Vekris[1], Frédéric Lang[2], Catalin Dima[1], Radu Mateescu[2]

[1] LACL, Université Paris-Est
61, avenue du Général de Gaulle, F-94010 Créteil, France
[2] Inria Grenoble Rhône-Alpes and LIG – CONVECS team
655, avenue de l'Europe, Montbonnot, F-38334 Saint Ismier, France

Abstract. EB^3 is a specification language for information systems. The core of the EB^3 language consists of process algebraic specifications describing the behaviour of the entities in a system, and attribute function definitions describing the entity attributes. The verification of EB^3 specifications against temporal properties is of great interest to users of EB^3. In this paper, we propose a translation from EB^3 to LOTOS NT (LNT for short), a value-passing concurrent language with classical process algebra features. Our translation ensures the one-to-one correspondence between states and transitions of the labelled transition systems corresponding to the EB^3 and LNT specifications. We automated this translation with the EB^{3}2LNT tool, thus equipping the EB^3 method with the functional verification features available in the CADP toolbox.

Keywords: Process Algebras, Information Systems, EB^3, LOTOS NT, Model Checking, Verification

1. Introduction

The EB^3 method [FSt03] is an event-based paradigm tailored for information systems (ISs). EB^3 has been used in the research projects SELKIS [MIL$^+$11] and EB^{3}SEC [JFG$^+$10], whose primary aim is the formal specification of ISs with security policies. In the EB^{3}SEC project, real banking industry case studies have been studied, describing interaction with brokers, customers and external financial systems. The SELKIS project deals with two case studies from the medical domain. The first one draws data records from medical imaging devices. The access to these records is done via web-based applications. The second one deals with availability and confidentiality issues for medical emergency units evolving in a great mountain range, like the Alps in that case.

A typical EB^3 specification defines entities, associations, and their respective attributes. The process algebraic nature of EB^3 enables the explicit definition of intra-entity constraints, making them easy for the IS designer to review and understand. Yet, its particular feature compared to classical process algebras, such

Correspondence and offprint requests to: Dimitris Vekris, Email: Dimitrios.Vekris@u-pec.fr

as CSP [Hoa78], lies in the use of *attribute functions*, a special kind of recursive functions evaluated on the system execution trace. Combined with guards, attribute functions facilitate the definition of complex inter-entity constraints involving the history of events. The use of attribute functions simplifies system understanding, enhances code modularity, and streamlines maintenance. However, given that ISs are complex systems involving data management and concurrency, a rigorous design process based on formal specification using EB^3 must be completed with effective formal verification features.

Existing attempts for verifying EB^3 specifications are based on translations from EB^3 to other formal methods equipped with verification capabilities. A first line of work [GFL05, GFL06] focused on devising translations from EB^3 attribute functions and processes to the B method [Abr05], which opened the way for proving invariant properties of EB^3 specifications using tools like Atelier B [Cl]. The B method is mainly used to verify safety properties over system specifications. A full-scale verification of B specifications is impossible under the classical B method. As a result, the B models of [GFL05, GFL06] were not suitable for verifying liveness properties.

On the other hand, it is known that temporal logic can deal both with safety and liveness properties. Hence, another approach concerned with the verification of temporal logic properties of EB^3 specifications by means of model checking techniques was taken. For this purpose, the formal description and verification of an IS case-study using six verification tools was undertaken in [FFC10, Cho10]. This study revealed the necessity of branching-time logics for accurately characterizing properties of ISs, and also the fact that process algebraic languages are suitable for describing the behaviour and synchronization of IS entities. However, no attempt of providing a systematic translation from EB^3 to a target language accepted as input by a model checker was made so far.

At first sight, given that EB^3 has structured operational semantics based on a labelled transition system (LTS) model, its translation to a process algebra may seem straightforward. However, this exercise proved to be rather complex, the main difficulty being to translate a history-based language (the control-flow depends on the entire system trace) to a process algebra with standard LTS semantics. On the other hand, the original trace-based semantics defined for finite-state systems in [FSt03] denoted Sem_{T} gives rise to unbounded memory models as the current trace is part of the current system state and, therefore, in the absence of good abstractions capable of reducing them to finite-state models, only bounded model-checking can be applied [BCC$^+$99]. This restriction is present in the original approach [FSt03] and the subsequent model-checking attempt [FFC10] even if all entities considered in the IS are finite.

To overcome this difficulty, [VD13] proposes a formal semantics for EB^3 that treats attribute functions as state variables, the so-called *attribute variables*. Intuitively, coding attribute functions as part of the system state is beneficial from a model-checking point of view as the new formalisation dispenses with the system trace. The derived memory semantics denoted Sem_{M} serves as the basis for applying a simulation strategy of attribute variables in LNT, a new generation process algebraic specification language inspired from E-LOTOS [Lot01]. In the same paper, Sem_{M} are shown to be equivalent to Sem_{T}. Again no complete translation of EB^3 to LNT is provided.

Contribution

1. Based on the efficient memory Sem_{M} given in [VD13], we propose a rigorous translation algorithm from EB^3 to LNT [CCG05]. As far as we know, this is the first attempt to provide a general translation from EB^3 to a classical value-passing process algebra. Since our primary objective was to provide temporal property verification features for EB^3, we focused our attention on LNT, which is one of the input languages accepted by the CADP verification toolbox [GLMS10], and hence is equipped with on-the-fly model checking for action-based, branching-time logics involving data. Another important ingredient of the translation was the multiway value-passing rendezvous of LNT, which enabled to obtain a one-to-one correspondence between the transitions of the two LTSs underlying the EB^3 and LNT descriptions, and hence to preserve strong bisimulation. The presence of array types and of usual programming language constructs (e.g., loops and conditionals) in LNT was also helpful for specifying the memory, the Kleene star-closure operators, and the EB^3 guarded expressions containing attribute function calls. At last, the constructed data types and pattern-matching mechanisms of LNT enabled a natural description of EB^3 data types and attribute functions.
2. We implemented our translation in the EB^{3}2LNT tool, thus making possible the analysis of EB^3 specifications using all the state-of-the-art features of the CADP toolbox, in particular the verification of data-based temporal properties expressed in MCL [MT08] using the on-the-fly model checker EVALUA-

TOR 4.0. The formal correctness proof of equivalence for EB^3 specifications and the corresponding LNT specifications can be found in [Vek14].

The paper is organized as follows:

- Section 2 presents the functional requirements of an IS and the related EB^3 specification that serves as the basis for introducing EB^3 's syntax in Section 3.
- Section 4 discusses the verification requirements that will be addressed in this paper and Section 5 explains why existing literature does not suffice to address them.
- Section 6 presents the memory semantics Sem_{M} of EB^3 and compares it in terms of efficiency to the standard traced-based semantics Sem_{T}.
- Section 7 gives an overview of the EB^3 and LNT languages.
- Section 8 introduces MCL briefly.
- Section 9 presents our translation from EB^3 to LNT, implemented by the EB^{3}2LNT translator.
- Section 10 shows how EB^{3}2LNT, CADP and MCL can be used for verifying the correctness requirements of an IS.
- Section 11 summarizes the results and draws up lines for future work.

2. Example

We present the syntax of EB^3 by way of an intuitive example. First, we give the functional requirements of a library management system:

R1. A book can be acquired by the library. It can be discarded, but only if it has not been lent.

R2. An individual must join the library in order to borrow a book.

R3. A member can relinquish library membership only when all his loans have been returned.

R4. A member cannot borrow more than the loan limit defined at the system level for all users.

The EB^3 specification intended to satisfy these requirements is presented in Figure 1. Note that process *main* is the parallel interleaving between m instances of process *book* executed an arbitrary but bounded number of times (Kleene Closure *) and p instances of process *member* executed an arbitrary but bounded number of times. Process *book* stands for a book acquisition "*Acquire* (*bId*)" followed by its eventual discard "*Discard* (*bId*)". The attribute function "*borrower* (T, *bId*)" looks for actions of the form "*Lend* (*mId*, *bId*)" or "*Return* (*bId*)" in the trace T and returns the current borrower of book *bId* denoted as "*borrower* (*front* (T), *bId*)" or $\perp$ if the book is not lent. In process *book*, action "*Discard* (*bId*)" is thus guarded to guarantee that book *bId* cannot be discarded if it is currently lent. Process *loan* stands for a book loan "*Lend* (*bId*, *mId*)" followed by an eventual return to the library "*Return* (*bId*)". The attribute function "*nbLoans* (T, *mId*)" inspects for actions of the form "*Lend* (*bId*, *mId*)" incrementing the current number of *mId*'s loans and inspects for actions of the form "*Return* (*bId*)" decrementing the current number of *mId*'s loans. In process *loan*, action "*Lend* (*bId*, *mId*)" is thus guarded to guarantee that book *bId* is not currently being lent and member *mId*'s loans is inferior to *NbLoans*. Process *member* stands for a member registration followed by an arbitrary but bounded number of loans and an eventual membership annulment.

3. The Language EB^3

The EB^3 method has been specially designed to specify the functional behaviour of ISs. A standard EB^3 specification comprises:

1. a class diagram representing entity types and associations for the IS being specified,
2. a process algebra specification, denoted by *main*, describing the IS, i.e., the valid traces of execution describing its behaviour,
3. a set of attribute function definitions, which are recursive functions on the system execution trace, and
4. input/output rules to specify outputs for input traces, or SQL expressions used to specify queries on the class diagram.

$BID = \{b_1, \ldots, b_m\}, MID = \{m_1, \ldots, m_p\}$
$book\,(bId : BID) =$
  $Acquire\,(bId)\ .\ (borrower\,(\mathsf{T},\ bId) = \bot) \Rightarrow Discard\,(bId)$
$loan\,(mId : MID,\ bId : BID) =$
  $(borrower\,(\mathsf{T},\ bId) = \bot) \wedge (nbLoans\,(\mathsf{T}, mId) < NbLoans) \Rightarrow$
    $Lend\,(bId,\ mId)\ .\ Return\,(bId)$
$member\,(mId : MID) =$
  $Register\,(mId)\ .\ (|||\, bId : BID : loan\,(mId, bId)^*)\ .\ Unregister\,(mId)$
$main =$
  $(|||\, bId : BID : book\,(bId)^*)\ |||\ (|||\, mId : MID : member\,(mId)^*)$

$nbLoans\,(\mathsf{T} : \mathcal{T}, mId : MID) : Nat_\bot\ =$
  match $last\,(\mathsf{T})$ **with**
    $\bot : \bot$
  | $Lend\,(bId, mId) :$
      $nbLoans\,(front\,(\mathsf{T}), mId) + 1$
  | $Register\,(mId) : 0$
  | $Unregister\,(mId) : \bot$
  | $Return\,(bId) :$
      if $mId = borrower\,(\mathsf{T}, bId)$ **then**
        $nbLoans\,(front\,(\mathsf{T}), mId) - 1$
      else $nbLoans\,(front\,(\mathsf{T}), mId)$ **end if**
  | $_ : nbLoans\,(front\,(\mathsf{T}), mId)$
  end match

$borrower\,(\mathsf{T} : \mathcal{T}, bId : BID) : MID_\bot\ =$
  match $last\,(\mathsf{T})$ **with**
    $\bot : \bot$
  | $Lend\,(bId, mId) : mId$
  | $Return\,(bId) : \bot$
  | $_ : borrower\,(front\,(\mathsf{T}), bId)$
  end match

Fig. 1. EB^3 specification of a library management system

$$
\begin{array}{rcl}
EB^3 & ::= & A_1; \ldots; A_n; S_1; \ldots; S_m \\
A & ::= & f\,(\mathsf{T} : \mathcal{T}, \overline{y} : \overline{T}) : T = \textbf{match}\ last\,(\mathsf{T})\ \textbf{with} \\
 & & \quad \bot : v_0 \mid \alpha_1\,(\overline{x_1}) : v_1 \mid \ \ldots\ \mid \alpha_q\,(\overline{x_q}) : v_q\ [\mid _ : v_{q+1}] \\
S & ::= & P\,(\overline{x}) = E \\
E & ::= & \lambda \mid \alpha\,(\overline{v}) \mid E_1.E_2 \mid E_1 \mid E_2 \mid {E_0}^* \mid E_1 \mid [\Delta] \mid E_2 \mid \mid x{:}V{:}E_0 \mid \\
 & & \mid [\Delta] \mid x{:}V{:}E_0 \mid C \Rightarrow E \mid P\,(\overline{v})
\end{array}
$$

Fig. 2. EB^3 syntax

We limit the presentation to the process algebra and the set of attribute functions. The EB^3 syntax is presented in Figure 2 and the EB^3 memory semantics Sem_{M} [VD13] are given in Figure 3 as a set of Rules named M_1 to M_{11}. Both figures are commented below. The equivalence of the original trace semantics Sem_{T} [FSt03] and Sem_{M} can be found in [VD13].

We write $x, y, x_1, x_2, \ldots$ for variables and $v, w, v_1, v_2, \ldots$ for data expressions over user-defined domains, such as integers, Booleans and more complex domains that we do not give formally, for conciseness. Let *Act* be a set of *actions* written $\rho, \rho_1, \rho_2, \ldots$ and *Lab* be a set of *labels* written $\alpha, \alpha_1, \alpha_2, \ldots$. Each action ρ is either the *internal action* written λ, or a *visible action* of the form "$\alpha\,(\overline{v})$", where $\alpha \in Lab$. Hence, *visible actions* consist of *labels* and data vectors. The notion of EB^3 actions coincides with the notion of CSP internal actions [Hoa78]. We also use the overlined notation as a shorthand notation for lists, e.g., $\overline{x}$ denotes a list of variables $x_1, \ldots, x_n$ of arbitrary length. An EB^3 specification consists of a set of attribute function definitions $A_1, \ldots, A_n$, and of a set of process definitions of the form "$P\,(\overline{x}) = E$", where P is a process name and E is a *process expression*.

Attribute functions. Attribute function definitions are denoted by the symbol A in the grammar of Figure 2. We assume a set of *labels* $Lab = \{\alpha_1, \ldots, \alpha_q\}$ (λ not included). Each α_j has formal parameters $\overline{x_j}$. Attribute functions $\{f_1, \ldots, f_n\}$ are defined recursively on the current trace T representing the history of actions with the aid of functions $last\,(\mathsf{T})$ which denotes the last action of the trace, and $front\,(\mathsf{T})$ which denotes the trace without its last action. The symbol $\bot$ represents the undefined value. In particular, both

$$(\mathrm{M}_1)\ \frac{}{\rho \xrightarrow{\rho} \surd} \qquad (\mathrm{M}_7)\ \frac{E_1 \xrightarrow{\rho} E_1' \qquad E_2 \xrightarrow{\rho} E_2'}{E_1\ |[\Delta]|\ E_2 \xrightarrow{\rho} E_1'\ |[\Delta]|\ E_2'}\ in\,(\rho,\Delta)$$

$$(\mathrm{M}_2)\ \frac{E_1 \xrightarrow{\rho} E_1'}{E_1.E_2 \xrightarrow{\rho} E_1'.E_2} \qquad (\mathrm{M}_8)\ \frac{E_1 \xrightarrow{\rho} E_1'}{E_1\ |[\Delta]|\ E_2 \xrightarrow{\rho} E_1'\ |[\Delta]|\ E_2}\ \neg in\,(\rho,\Delta)$$

$$(\mathrm{M}_3)\ \frac{E_2 \xrightarrow{\rho} E_2'}{\surd.E_2 \xrightarrow{\rho} E_2'} \qquad (\mathrm{M}_9)\ \frac{}{\surd\ |[\Delta]|\ \surd \xrightarrow{\lambda} \surd}$$

$$(\mathrm{M}_4)\ \frac{E_1 \xrightarrow{\rho} E_1'}{E_1 \mid E_2 \xrightarrow{\rho} E_1'} \qquad (\mathrm{M}_{10})\ \frac{E_0 \xrightarrow{\rho} E_0'}{C \Rightarrow E_0 \xrightarrow{\rho} E_0'}\ \|C\ [\ f\,(\mathsf{T},\overline{v}) := \mathsf{f}[\,\overline{v}\,]\ ]\|$$

$$(\mathrm{M}_5)\ \frac{}{{E_0}^* \xrightarrow{\lambda} \surd} \qquad (\mathrm{M}_{11})\ \frac{E[\overline{x} := \overline{v}] \xrightarrow{\rho} E'}{P\,(\overline{v}) \xrightarrow{\rho} E'}\ P\,(\overline{x}) = E$$

$$(\mathrm{M}_6)\ \frac{E_0 \xrightarrow{\rho} E_0'}{{E_0}^* \xrightarrow{\rho} E_0'.{E_0}^*}$$

Fig. 3. EB^3 memory semantics Sem_{M}

last (T) and *front* (T) match $\perp$ when the trace is empty. The symbol _ (wildcard) matches all actions not matched by any of the preceding action patterns $\alpha_1\,(\overline{x_1}), \ldots, \alpha_q\,(\overline{x_q})$.

Note that the attribute functions f_i share the same vector of formal parameters $\overline{y}$, which we call *attribute vector*[1], modulo renaming, and, therefore, have the same arity, which is by no means restrictive in terms of language expressiveness. Each v_i for $i \in \{0, \ldots, n\}$ is an expression of the same type as f's return type built over the variables $\overline{y} \cup \overline{x_i}$. In real EB^3 specifications with distinct formal parameters $\overline{y_i}$ each corresponding to every f_i, $\overline{y}$ can be formalized as the union of the $\overline{y_i}, i \in 1..n$. We also assume that the attribute functions are ordered, so that for all $h \in 1..n, i \in 1..n, j \in 1..q$, every function call of the form "$f_h\,(\mathsf{T}, \ldots)$" occurring in w_i^j satisfies $h < i$ and every call of the form "$f_h\,(\mathit{front}\,(\mathsf{T}), \ldots)$" satisfies $h \geq 0$. Such an ordering can be constructed if the EB^3 specification does not contain circular dependencies between function calls, which would potentially lead to infinite *attribute function* evaluation. In particular, the definition of an attribute function f_i cannot contain recursive calls of the form "$f_i\,(\mathsf{T}, \ldots)$", but only recursive calls of the form "$f_i\,(\mathit{front}\,(\mathsf{T}), \ldots)$". Note that this does not limit the expressiveness of EB^3 *attribute functions*, because every recursive computation on data expressions only (which keeps the trace unchanged) can be described using standard functions and not attribute functions. The EB^3 specification of Figure 1 satisfies this ordering, as the definition of *borrower* contains calls of the form "*borrower* (*front* (T), *bId*)", while the definition of *nbLoans* contains calls of the form "*nbLoans* (*front* (T), *mId*)" and "*borrower* $(\mathsf{T}$, *bId*)".

Process expressions. Expressions are built over variables, constants, and standard operations. An action ρ is the simplest process expression, whose semantics are given by Rule M_1. The symbol $\surd$ (which is not part of the user syntax) denotes successful execution. The *trace* T of an EB^3 specification at a given moment consists of the sequence of visible actions executed since the start of the system. (Note therefore that λ does not appear in the trace.) At system start, the trace is empty. If T denotes the current trace and action ρ can be executed, then $\mathsf{T}.\rho$ denotes the trace just after executing ρ.

EB^3 processes can be combined with classical process algebra operators such as the *sequence* "$E_1.E_2$" $(\mathrm{M}_2, \mathrm{M}_3)$, the *choice* "$E_1 \mid E_2$" (M_4) and the *Kleene* closure "${E_0}^*$" $(\mathrm{M}_5, \mathrm{M}_6)$. Rules ($\mathrm{M}_7$ to M_9) define *parallel* composition "$E_1\ |[\Delta]|\ E_2$" of E_1, E_2 with synchronization on $\Delta \subseteq \mathit{Lab}$. The condition "$in\,(\rho, \Delta)$" is true iff the label of ρ belongs to Δ. The symmetric Rules for choice and parallel composition have been omitted for brevity. Expressions "$E_1\ |||\ E_2$" and "$E_1\ ||\ E_2$" are equivalent respectively to "$E_1\ |[\emptyset]|\ E_2$" and "$E_1\ |[\mathit{Lab}]|\ E_2$".

[1] each $y \in \overline{y}$ is called *attribute parameter* correspondingly

Quantification is permitted for *choice* and *parallel* composition. If V is a set of expressions $\{v_1, \ldots, v_n\}$, "$|\, x : V : E_0$" and "$|[\Delta]|\, x : V : E_0$" stand respectively for "$E_0[x := v_1] \mid \ldots \mid E_0[x := v_n]$" and "$E_0[x := v_1]\ |[\Delta]|\ \ldots\ |[\Delta]|\ E_0[x := v_n]$", where "$E[x := v]$" denotes the replacement of all occurrences of x by v in E. For instance, "$||\, x : \{1, 2, 3\} : a\,(x)$" stands for "$a\,(1) \,||\, a\,(2) \,||\, a\,(3)$".

At last, named processes can be instantiated as usual (M_{11}). Given an EB^3 process expression E, we write $vars\,(E)$ for the set of variables occurring free in E.

4. Verification Requirements

We need to verify that the EB^3 specification of Figure 1 satisfies the functional requirements of the simplified library management system presented in Section 2. To this end, we divide them into *liveness* and *safety* properties. *Liveness* properties express the possibility that a certain event will take place and *safety* properties express the certitude that a certain event will never take place. Requirements R1 and R3 are typical *liveness* properties and requirements R2 and R4 are typical *safety* properties.

5. Related Works

Following the technique of [GFL05, GFL06], one may devise a translation from the EB^3 specification of Figure 1 to the B method. This approach suffices to verify requirements R2 and R4, since *safety* properties can be formalized in the B method, but the verification of requirements R1 and R3 cannot be addressed by way of classical refinement techniques. An important class of liveness properties can be verified in Event-B [Abr10] as explained in [HA11, Gro06]. In [HA11], the system is defined as an Event-B model and the *liveness* property ϕ in question is expressed in Linear Tree Logic (LTL) [Pnu77]. Then, the Büchi automaton corresponding to ϕ is translated to an equivalent Event-B model that is added to the initial Event-B model. Finally, the consistency of the final model implies the satisfaction of ϕ. In [Gro06], the *liveness* property ϕ is expressed as proof obligations of the Event-B model, whose resolution implies the satisfaction of ϕ. More recent work [STW$^+$14] in the field sets a framework for preserving liveness and fairness properties in LTL along successive refinement steps of Event-B machines. The refinement of events is allowed among the possible refinement steps of this scheme.

However, the use of LTL is usually inadequate for the verification of *liveness* properties over ISs. The reason lies in the presence of loops in the IS specification leading to infinite paths. To see this, let's consider a certain class of liveness properties called reachability properties. Reachability properties can be expressed in Computational Tree Logic (CTL) [CES86] formulas "$AG(\psi \Rightarrow EF\phi)$" that are satisfied if for all states that are reachable from the initial states of the system, where formula ψ is valid, there is a sequence of transitions that lead to states of the system, where formula ϕ is valid. Reachability properties cannot be expressed and verified in LTL. Instead, only branching-time logics like CTL can be used for this purpose. Hence, the verification techniques of [HA11, Gro06, STW$^+$14] are not applicable here.

In [ETL$^+$04], the authors propose the translation of EB^3 specifications to equivalent CSP‖B [ST05, TSB03] specifications. CSP‖B is an hybrid method that combines features of CSP and the B method. The dynamic behaviour of the IS in question is directly cast to a CSP specification. CSP shares common process algebraic features with EB^3. The system state inherent to the attribute function definitions has to be defined explicitly via the state variables of a B specification. Then, every CSP action execution triggers a unique operation of the B specification that has an immediate effect on the state variables. The properties over the CSP‖B specification have to be expressed in CSP and the verification is carried out by the FDR2 model checker [Fdr97]. *Liveness* properties expressed as temporal properties can be cast to FDR2 as explained in [LMC00]. As a result, both *liveness* and *safety* properties of the library specification can be verified with the aid of refinement checking and the FDR2 model checker. Although the method of [ETL$^+$04] is systematic, the translation of EB^3 specifications to equivalent CSP‖B models is not automatic, but still necessitates user intervention.

On the other hand, it is known that temporal logic can deal both with *safety* and *liveness* properties. Hence, another approach concerned with the verification of temporal logic properties of EB^3 specifications by means of model checking techniques is taken. For this purpose, the formal description and verification of the library management system specification [Ger06] using six verification tools are undertaken in [FFC10, Cho10] namely SPIN [Hol04], NuSMV [CCG$^+$02], FDR2, CADP, ALLOY [Jac06] and ProB [LB03]. SPIN,

CADP and FDR2 are called explicit model checkers. In CADP and FDR2, the transition system describing the system specification is constructed explicitly prior to the property verification, whereas in the case of SPIN, the property is verified while the transition system is constructed in parallel (on-the-fly verification). Nu-SMV belongs to a group of model-checkers called symbolic model checkers. In symbolic model checking language specifications, the transition system is given in the form of Boolean formula. ALLOY is a language based on first-order logic. Moreover, ALLOY models are defined purely as sets and relations on these sets. Properties are expressed as boolean expressions and verification is reduced to the SAT problem. It is worth mentioning that ALLOY's language used to model the system is the same as the language used to specify the property we need to verify. The same applies to PROB, whereas Spin, CADP, NuSMV use either action-based or state-based temporal logics.

In the case of ALLOY and the case of reachability properties, the authors of [FFC10] specified (manually) ad hoc traces leading to the states of the system model satisfying these properties, which is the reason why ALLOY was proven to be the most efficient among the six verification tools used in the paper. The drawback of this approach is that it necessitates good knowledge of the underlying transition system by the user. As a whole, the study in [FFC10] revealed the necessity of branching-time logics for accurately characterizing properties of ISs, and the fact that process algebraic languages are suitable for describing the behaviour and synchronization of IS entities. However, no attempt of providing a systematic translation from EB^3 to a target language accepted as input by a model checker is made so far.

In [MF15], the verification of reachability properties "$AG(\psi \Rightarrow EF\phi)$" with the aid of the B method is undertaken. The goal of this study is to construct a program p that *refines* a reachability property q in the sense that the set of execution paths related to program p is strictly a subset of the execution paths related to the satisfaction of reachability formula q. The construction of p follows the algorithmic refinement laws of Morgan [Mor98] that are intended to decompose q into a sequence of properties, which can be trivially refined by simple system transitions (actions). This method is not automatic, as the user is asked to specify the program that refines the given specification.

In this article, we address the problem of automatic translation (no user intervention) of EB^3 models to equivalent LNT models knowing that:

- LNT is one of the input languages accepted by the CADP verification toolbox featuring full-blown temporal property validation capabilities, and that
- the model checking on LNT models allows branching-time reasoning, hence allowing the verification of *liveness* properties on IS specifications.

6. Memory Semantics

We define the memory semantics Sem_{M} [VD13] and demonstrate by way of example why Sem_{M} surpasses the original trace-based memory Sem_{T} [FSt03] in terms of efficiency. For defining the Sem_{M} semantics for attribute functions, attribute functions are turned into state variables (the memory $\mathsf{M} = \{\mathsf{f_1}, \ldots, \mathsf{f_n}\}$) carrying the effect of the system trace on their corresponding values. This avoids keeping the (ever-growing) trace leading to a finite state model. If $f_i\,(\mathsf{T}, x_1{:}T_1, \ldots, x_l{:}T_l)$ is an attribute function, we construct $|T_1| \times \ldots \times |T_l|$ state variables denoted by $\mathsf{f_i}$, where $|T_i|$ $(i \in 1..l)$ stands for T_i's cardinality. Then, f'_i denoting $\mathsf{f_i}$'s update upon execution of $\alpha_j\,(\overline{v_j})$ is given as follows:

$$\mathsf{M}' = \{\mathsf{f}'_1, \ldots, \mathsf{f}'_n\} \tag{1}$$

$$\mathsf{f}'_i\,[\,\overline{x}\,] = \|\ w_i^j\ [\ f_h\,(\mathsf{T}, \overline{v_h}) \leftarrow \mathsf{f_h}'[\,\overline{v_h}\,]\ \mid\ h < i\ ] \tag{2}$$

$$[\ f_h\,(\mathit{front}\,(\mathsf{T}), \overline{v_h}) \leftarrow \mathsf{f_h}[\,\overline{v_h}\,]\ \mid h \geq 0\ ]\ \|\ (\overline{x}) \text{ for } j > 0 \tag{3}$$

where the notation $\|.\|$ denotes the evaluation of data expressions based on the classic interpretations for Peano Arithmetic, Set Theory and Boolean Logic under the current memory M (implicit both in Figure 3 and above) carries the values of attribute variables. If action $\alpha_j\,(\overline{v_j})$ can be executed, then M' denotes the memory after its execution. By definition, the inert action λ has no effect on the memory M, i.e. $\mathsf{M}' = \mathsf{M}$. The memory M is successively updated from $\mathsf{f}_1, .., \mathsf{f}_n$ to $\mathsf{M}' = \{\mathsf{f_1}', \ldots, \mathsf{f_n}'\}$ as depicted in Definition (3). Calls of the form $f_h\,(\mathsf{T}, \overline{v_h})$ within w_i^j are replaced by already-updated $\mathsf{f_h}'[\,\overline{v_h}\,]$, since f_h takes T as argument for $h < i$. Calls of the form $f_h\,(\mathsf{T}, \overline{v_h})$ are replaced by the value of $\mathsf{f_h}[\,\overline{v_h}\,]$, since f_h takes $\mathit{front}\,(\mathsf{T})$ as argument for $h \geq 0$. This computation caters for the complete discharge of the trace T from w_i^j, while consistently

Memory				
State	$\mathsf{borrower}[\mathsf{b}_1]$	$\mathsf{borrower}[\mathsf{b}_2]$	$\mathsf{nbLoans}[\mathsf{m}_1]$	$\mathsf{nbLoans}[\mathsf{m}_2]$
A	$\bot$	$\bot$	$\bot$	$\bot$
A'	$\bot$	$\bot$	$\bot$	$\bot$
B	$\bot$	$\bot$	$\bot$	$\bot$
B'	$\bot$	$\bot$	$\bot$	0
C	$\bot$	$\bot$	0	0
D	m_1	$\bot$	1	0

Fig. 4. Evolution of *Atrribute variables*

updating the memory M. We denote by $\mathsf{M}_0 = \{\mathsf{f}_1^0, \ldots, \mathsf{f}_n^0\}$ the memory at system start, which corresponds to "$\mathsf{T} = [\,]$" and "$j = 0$" and set $\mathsf{f}_i^0[\overline{x}] = \bot$ for every parameter vector $\overline{x}$ and every $i \in \{1, \ldots, n\}$.

The *guarded expression* process "$C \Rightarrow E_0$" (M_{10}) can execute E_0 if the Boolean condition C holds under memory M, which is denoted by the side condition "$\|C\ [\ f_i\,(\mathsf{T}, \overline{v}) := \mathsf{f}_i[\,\overline{v}\,]\ ]\|$" such that "$\mathsf{M} = \{\mathsf{f}_1, \ldots, \mathsf{f}_n\}$". Since C may contain calls to attribute functions of the form $f\,(\mathsf{T}, \overline{v})$, the evaluation of C necessitates their replacement by their corresponding attribute variables $\mathsf{f}[\,\overline{v}\,]$. The trace T is discharged from C and, hence, the evaluation of the guard C does not depend on T. Note that the evaluation of C and the execution of the first action ρ in E_0 are simultaneous, i.e., no action is allowed in concurrent processes in the meantime. We call this property the *guard-action atomicity*. This property is essential for consistency as, by side effects, the occurrence of actions in concurrent processes could implicitly change the value of C before the guarded action has been executed.

Execution. We demonstrate how the memory semantics Sem_{M} works for the simplified library management system, whose specification (processes and attribute functions) in EB^3 is given in Figure 1. As an example, we set $m = p = NbLoans = 2$, i.e. we consider two books b_1 and b_2, and two members m_1 and m_2. The memory has four cells:

$$\mathsf{M} = (borrower[b_1], borrower[b_2], nbLoans[m_1], nbLoans[m_2]).$$

The first two cells keep the two values of the attribute function $borrower\,(\mathsf{T}, \bullet)$ for a given trace T, and the last two keep the values of $nbLoans\,(\mathsf{T}, \bullet)$. After every step, the new value of each cell can be calculated from the previous memory and the action that has just been executed. The memory is initially set to $(\bot, \bot, \bot, \bot)$. After the trace "$Acquire\,(b_1).Acquire\,(b_2).Register\,(m_1).Register\,(m_2)$" the memory contains $(\bot, \bot, 0, 0)$. If the action "$Lend\,(b_1, m_1)$" is then executed, the new memory is $(m_1, \bot, 1, 0)$. For instance, the new value m_1 for $borrower[b_1]$ is obtained from the Rule "$Lend\,(bId, mId) : mId$" in the definition of the attribute function *borrower* (see Fig. 1), and the new value 1 for $nbLoans[m_1]$ by the Rule "$Lend\,(bId, mId) : nbLoans\,(front\,(\mathsf{T}), mId) + 1$" of the attribute function *nbLoans*, where the value of $nbLoans\,(front\,(\mathsf{T}), m_1)$ corresponds to the value of $nbLoans[m_1]$ in the previous memory state (value 0). The evolution of *attribute variables* is depicted in Figure 4, where (A') is the state after the execution of $Acquire\,(b_1)$, (B) is the state the moment succeeding the execution of $Acquire\,(b_2)$, (B') is the state the moment succeeding the execution of $Register\,(m_1)$, (C) is the state the moment succeeding the execution of $Register\,(m_2)$ and (D) is the state the moment succeeding the execution of $Lend\,(b_1, m_1)$.

Figure 5 depicts the evolution of process expression *main* describing the library management system in states (A), (B), (C), and (D). Transition (C)→(D) entails the evaluation of the following guard:

$$(borrower\,(\mathsf{T}, b_1) = \bot)\ \wedge\ (nbLoans\,(\mathsf{T}, m_1) < 2) \tag{4}$$

where T corresponds to the system trace at state (C), i.e. $\mathsf{T} = \mathsf{T}_C$. Condition (4) illustrates the conditions under which member m_1 can lend book b_1 (notably if the book is available and the number of loans carried out by m_1 is inferior to two). We concentrate on the evaluation of the *attribute function* call "$borrower\,(\mathsf{T}, b_1)$" w.r.t. Sem_{M} and then we compare it with the evaluation of "$borrower\,(\mathsf{T}, b_1)$" w.r.t. the

$main$ (A)

$\xrightarrow{Acquire\,(b_2).\ Acquire\,(b_1)}$

$borrower\,(\mathsf{T}_B, b_1) = \bot \rightarrow Discard\,(b_1).\ book\,(b_1)^*\ |||$
$borrower\,(\mathsf{T}_B, b_2) = \bot \rightarrow Discard\,(b_2).\ book\,(b_2)^*\ |||$
$(\ ||| \ mId : MID : member\,(mId)^*\ )$ (B)

$\xrightarrow{Register\,(m_2).\ Register\,(m_1)}$

$borrower\,(\mathsf{T}_C, b_1) = \bot \rightarrow Discard\,(b_1).\ book\,(b_1)^*\ |||$
$borrower\,(\mathsf{T}_C, b_2) = \bot \rightarrow Discard\,(b_2).\ book\,(b_2)^*\ |||$
$(\ ||| \ bId : BID : loan\,(m_1, bId)^*\ ).\ Unregister\,(m_1).\ member\,(m_1)^*\ |||$
$(\ ||| \ bId : BID : loan\,(m_2, bId)^*\ ).\ Unregister\,(m_1).\ member\,(m_2)^*$ (C)

$\xrightarrow{Lend\,(b_1,\ m_1)}$

$borrower\,(\mathsf{T}_D, b_1) = \bot \rightarrow Discard\,(b_1).\ book\,(b_1)^*\ |||$
$borrower\,(\mathsf{T}_D, b_2) = \bot \rightarrow Discard\,(b_2).\ book\,(b_2)^*\ |||$
$(Return(b_1).\ loan(m_1, b_1)^*\ ||| \ loan(m_1, b_2)^*).\ Unregister\,(m_1).\ member\,(m_1)^*\ |||$
$(\ ||| \ bId : BID : loan\,(m_2, bId)^*).\ Unregister\,(m_1).\ member\,(m_2)^*$ (D)

Fig. 5. Sample execution

classical traced-based memory Sem_{T}. Following (3) and the definition of *borrower* in Figure 1, *attribute function* call "$borrower\,(\mathsf{T}, b_1)$" is evaluated w.r.t. Sem_{M} as follows:

$$\| \ borrower(\mathsf{T}_C, b_1)\ \| = borrower_{B'}[b_1] = \bot$$

where "$borrower_{B'}[b_1]$" is the (stocked) *attribute variable* at state (B'). Following the *attribute function* definition of *borrower* in Figure 1, the evaluation of "$borrower\,(\mathsf{T}_C, b_1)$" w.r.t. Sem_{T} (the standard ML-like evaluation of recursive functions denoted as $\|.\|_{\mathsf{T}}$ to mark the difference with the $\|.\|$ evaluation) triggers the complete traversal of the whole trace as demonstrated below:

$$\begin{aligned}
\| \ borrower(\mathsf{T}_C, b_1)\ \|_{\mathsf{T}} &= \| \ borrower\,(front\ (\mathsf{T}_C),\ b_1)\ \|_{\mathsf{T}} \\
&= \| \ borrower\,(\mathsf{T}_{B'},\ b_1)\ \|_{\mathsf{T}} \\
&= \| \ borrower\,(front\ (\mathsf{T}_{B'}),\ b_1)\ \|_{\mathsf{T}} \\
&= \| \ borrower\,(\mathsf{T}_B,\ b_1)\ \|_{\mathsf{T}} \\
&= \| \ borrower\,(front\ (\mathsf{T}_B),\ b_1)\ \|_{\mathsf{T}} \\
&= \| \ borrower\,(\mathsf{T}_{A'},\ b_1)\ \|_{\mathsf{T}} \\
&= \| \ borrower\,(front\ (\mathsf{T}_{A'}),\ b_1)\ \|_{\mathsf{T}} \\
&= \| \ borrower\,(\mathsf{T}_A,\ b_1)\ \|_{\mathsf{T}} \\
&= \| \ \bot\ \|_{\mathsf{T}} = \bot
\end{aligned}$$

from which follows directly that Sem_{M} is computationally more efficient than Sem_{T}.

$$
\begin{array}{lcl}
B & ::= & \textbf{stop} \mid \textbf{null} \mid G\,(O_1, \ldots, O_n)\ \textbf{where}\ E \mid B_1; B_2 \\
 & \mid & \textbf{if}\ E\ \textbf{then}\ B_1\ \textbf{else}\ B_2\ \textbf{end if} \mid \textbf{var}\ x{:}T\ \textbf{in}\ B\ \textbf{end var} \mid x := E \mid \\
 & \mid & \textbf{loop}\ L\ \textbf{in}\ B\ \textbf{end loop} \mid \textbf{break}\ L \mid \textbf{select}\ B_1\ \texttt{[]}\ \ldots\ \texttt{[]}\ B_n\ \textbf{end select} \\
 & \mid & \textbf{par}\ G_1, \ldots, G_m\ \textbf{in}\ B_1\ \texttt{||}\ \ldots\ \texttt{||}\ B_n\ \textbf{end par} \mid P\,[G_1, \ldots, G_n]\ (E_1, \ldots, E_n) \\
O & ::= & \texttt{!}E \mid \texttt{?}x
\end{array}
$$

Fig. 6. LNT syntax (limited to the fragment used in this paper)

7. The Language LNT

LNT aims at providing the best features of imperative and functional programming languages and value-passing process algebras. It has a user friendly syntax and formal operational semantics defined in terms of labeled transition systems (LTSs). LNT is supported by the LNT.OPEN tool of CADP, which allows the on-the-fly exploration of the LTS corresponding to a LNT specification.

We present the fragment of LNT that serves as the target of our translation. LNT terms B also called as behaviours are built from actions, choice (**select**), conditional (**if**), sequential composition (**;**), breakable loop (**loop** and **break**) and parallel composition (**par**). Communication is carried out by rendezvous on gates, written G, G_1, ..., G_m, and may be guarded using Boolean conditions on the received values (**where** clause). LNT allows multiway rendezvous with bidirectional (send/receive) value exchange on the same gate occurrence, each offer O being either a send offer (`!`) or a receive offer (`?`), independently of the other offers. Expressions E are built from variables, type constructors, function applications and constants. Labels L identify loops, which can be escaped using "**break** L" from inside the loop body. Processes are parameterized by gates and data variables. LNT syntax and semantics are formally defined in SOS style in [CCG05].

8. MCL

MCL is an extension of the alternation-free modal μ-calculus [EL$^+$86] with action predicates enabling value extraction, modalities containing extended regular expressions on transition sequences, quantified variables and parameterized fixed point operators, programming language constructs, and fairness operators encoding generalized Büchi automata. These features make possible a concise and intuitive description of safety, liveness, and fairness properties involving data, without sacrificing the efficiency of on-the-fly model checking, which has a linear-time complexity for the dataless MCL formulas [MT08].

In particular, MCL consists of data expressions e, action formulas α and state formulas ϕ. Let a set of data variables $\mathcal{X}$ and a set of function identifiers $\mathcal{F}$ with standard interpretation. Data expressions e are defined as follows:

$$e ::= x \quad | \quad f(e_1, \ldots, e_n),$$

where $x \in \mathcal{X}$ and $f \in \mathcal{F}$ denoting typed function identifiers.

Action formulas α consist of:

- action patterns of the form $\{c\ \texttt{!}e_1\ \ldots\ \texttt{!}e_n\}$ such that, if action $\{c\ v_1\ \ldots\ v_n\}$ occurs, v_k matches expression e_k for $k \in \{1, \ldots, n\}$,
- action patterns of the form $\{c\ \texttt{?}x_1 : T_1\ \ldots\ \texttt{?}x_n : T_n\}$[2] such that, if action $\{c\ v_1\ \ldots\ v_n\}$ occurs, $\mathsf{v_k}$ is assigned to variable $\mathsf{x_k}$ for $k \in \{1, \ldots, n\}$, and
- usual Boolean operators.

To sum up, the abstract syntax of action formulas α is given by the following grammar:

$$\alpha ::= \{c\ \texttt{!}e_1\ \ldots\ \texttt{!}e_n\} \quad | \quad \{c\ \texttt{?}x_1 : T_1\ \ldots\ \texttt{?}x_n : T_n\} \quad | \quad \neg\alpha \quad | \quad \alpha_1 \wedge \alpha_2 \quad | \quad \alpha^*$$

Let propositional variables $Y \in \mathcal{Y}$ that denote functions $F : T_1 \times \ldots \times T_n \rightarrow 2^S \in \mathcal{F}$, where $T_1, \ldots, T_n$ are domains and S is the state space. State formulas consist of data expressions, propositional variables $Y \in \mathcal{Y}$, boolean operators, quantifiers and fixed point operators.

[2] T_k denotes x_k's type as well as the corresponding domain of elements that can be assigned to x_k

Note also that fixed point operations fall within the scope of an even number of negations (MCL formulas $\mu(\ldots)Y.\phi$ and $\nu(\ldots)Y.\phi$ are syntactically monotonic) [Koz83]. For the efficiency of model checking, mutual recursion between minimal and maximal fixed point operators is not allowed [EL$^+$86].

We assign values to propositional variables $x \in \mathcal{X}$ appearing in expressions e and action formulas α that are denoted as $\delta : \mathcal{X} \rightarrow T_1 \cup \ldots \cup T_n$. Moreover, we assign functions to all free propositional variables Y appearing in MCL state formulas that are denoted as $\rho : \mathcal{Y} \rightarrow \mathcal{F}$. Notations "$\mu Y.\phi$" and "$\nu Y.\phi$" denote the corresponding fixed points of monotonic functions over $Y \rightarrow 2^S$.

Other useful operators like **if** and **case** can also be used to construct MCL formulas. Their definition is found in [MT08].

Action patterns are enriched with additional features such as the wilcard clause **any** that matches any value and the "**where** V" clause denoting that the pattern-matching takes place on condition that condition V evaluates to true. Remark that parameters $?x : T$ can appear syntactically in V.

The abstract syntax of MCL state formulas is given by the following grammar:

$$\begin{array}{rl} \phi ::= & e \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \langle\, \alpha \,\rangle\, \phi \mid \mu Y.\ \phi \\ \mid & \textbf{exists } x_1 : T_1, \ldots, x_n : T_n.\ \phi \mid Y(e_1, \ldots, e_n) \\ \mid & \textbf{let } x_1 : T_1 := e_1, \ldots, x_n : T_n := e_n \textbf{ in } \phi \textbf{ end let} \\ \mid & \textbf{if } \phi_1 \textbf{ then } \phi'_1 \textbf{ elsif } \phi_2 \textbf{ then } \phi'_2 \textbf{ else } \phi'_3 \textbf{ end if} \\ \mid & \textbf{case } e \textbf{ is } p_1 \rightarrow \phi_1 \mid \ldots \mid p_n \rightarrow \phi_n \textbf{ end case} \end{array}$$

The necessity modality is the dual of possibility modality "$[\,\alpha\,]\ \phi = \neg\langle\,\alpha\,\rangle\ \neg\phi$". The maximal fixed point operator is the dual of the minimal fixed point operator: "$\nu(\ldots)Y.\phi = \neg\mu(\ldots)Y.\neg\phi[Y/\neg Y]$", where $[Y/\neg Y]$ denotes the syntactic substitution of Y by $\neg Y$. Note that "$\langle\,\alpha^*\,\rangle\ \phi$" is syntactic sugar for "$\mu Y(\phi \vee \langle\,\beta\,\rangle\ Y)$".

MCL's expressiveness is not limited to *safety* and *liveness* properties. Deadlock freedom is formalized as "$[\,true^*\,]\ \langle\,true\,\rangle\ true$". The fair reachability [QS83] of an action a is formalized as "$[\,(\neg a)^*\,]\ \langle\,(\neg a)^*.a\,\rangle\ true$". For more complex fairness properties, we use the infinite looping operator $\Delta\beta$ of PDL-Δ [Str82], denoted as $\beta@$ in MCL, which states the existence of an infinite (unfair) sequence made by concatenating subsequences satisfying β.

Let's consider the following MCL example:

$$[\,\textbf{true}^*\,.\,\{\texttt{RESERVE ?}M : \textbf{string ?}B : \textbf{string}\}\,.\,(\textbf{not } (\{\texttt{TAKE !}M\ \texttt{!}B\}))^*\,.\,\{\texttt{RESERVE !}M\ \texttt{!}B\}\,]\ \textbf{false}$$

The expression inside $[\,.\,]$ describes the set of execution paths starting with a bounded number of actions (denoted as $\textbf{true}^*$) followed by a `RESERVE` action of book B by member M, a bounded number of actions that are not `TAKE` actions of B by M and a `RESERVE` action of book B by member M. Finally, **false** notation is used to denote that the paths described inside $[\,.\,]$ are not possible (*safety* property).

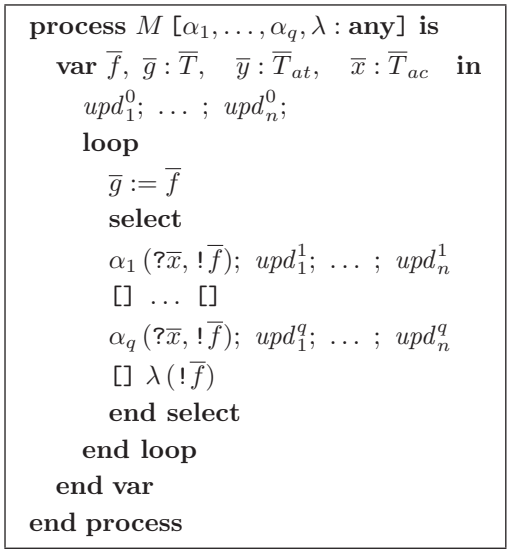
9. Translation of EB^3 to LNT

9.1. Translation of attribute functions.

Ordering attribute functions as described in Section 6 allows the memory to be updated consistently, from f_1 to f_n in turn. At every instant, already-updated values correspond to calls of the form $f_h\,(\mathsf{T}, \ldots)$ (the value of f_h on the current trace), whereas calls of the form $f_h\,(\mathit{front}\,(\mathsf{T}), \ldots)$ are replaced by accesses to a copy $\overline{g}$ of the memory $\overline{f}$, which was made before starting the update. This encoding thus enables the trace parameter to be discharged from function calls, ensuring that while updating f_i, accesses to f_h with $h < i$ necessarily correspond to calls with parameter T.

The process M is defined in Figure 7. It runs an infinite loop, which "listens" to all possible actions α_j of the system. At this point, we recall the existence of unique *attribute function* definition $f_i\ (\mathsf{T} : \mathcal{T}, y_1 : T_1, \ldots, y_s : T_s) : T$ (see Figure 2 for details). Each attribute variable f_i is an array with s dimensions, where s is the common arity for attribute functions f_i minus 1, because the trace parameter is now discharged. Each dimension of array f_i, thus, corresponds to one formal parameter in $\overline{y_i}$, so that:

$$f_i\,[\textbf{ord}\,(v_1)]\,\ldots\,[\textbf{ord}\,(v_s)]$$



```
process M [α_1, ..., α_q, λ : any] is
  var f, g : T,   y : T_at,   x : T_ac   in
    upd_1^0; ... ; upd_n^0;
    loop
      g := f
      select
      α_1 (?x, !f); upd_1^1; ... ; upd_n^1
      [] ... []
      α_q (?x, !f); upd_1^q; ... ; upd_n^q
      [] λ (!f)
      end select
    end loop
  end var
end process
```

Fig. 7. LNT code for the memory process implementing the attribute functions

$$
\begin{array}{l}
upd_i^j \doteq enum\,(\overline{y}, f_i\,[ord\,(\overline{y})] := mod\,(w_i^j)) \\
enum\,([\,], B) \doteq B \\
enum\,(y_k :: \overline{z}, B) \doteq y_k := first_k; \\
\qquad\qquad\qquad\qquad \textbf{loop } L_k \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad enum\,(\overline{z}, B) \\
\qquad\qquad\qquad\qquad\quad \textbf{if } y_k \neq last_k \textbf{ then } y_k := next_k\,(y_k) \textbf{ else break } L_k \textbf{ end if} \\
\qquad\qquad\qquad\qquad \textbf{end loop} \\
mod\,(w_i^0) \doteq w_i^0 \;\; [\; f_h\,(\mathsf{T},\ \overline{y}) := f_h\,[\textbf{ord}\,(\overline{y})] \quad | \quad \forall h < n \;] \\
\qquad\qquad\qquad\quad [\; f_h\,(front\,(\mathsf{T}),\ \overline{y}) := \bot \quad | \quad \forall h \in \{1, \ldots, n\} \;] \\
mod\,(w_i^j) = w_i^j \;\; [\; f_h\,(\mathsf{T},\ \overline{y}) := f_h\,[\textbf{ord}\,(\overline{y})] \quad | \quad \forall h < n \;] \\
\qquad\qquad\qquad\quad [\; f_h\,(front\,(\mathsf{T}),\ \overline{y}) := g_h\,[\textbf{ord}\,(\overline{y})] \quad | \quad \forall h \in \{1, \ldots, n\} \;], \quad \text{if } j \neq 0
\end{array}
$$

Fig. 8. LNT code for expression upd_i^j of Figure 7

encodes the current value of:

$$f_i\,(\mathsf{T}, v_1, \ldots, v_s),$$

where **ord** (v_k) is a predefined LNT function that denotes the *ordinate* of the value that corresponds to variable v_k, i.e., a unique number between 1 and the cardinal number $|D_k|$ of domain D_k that stocks elements of type T_k.

Expression upd_i^j for $i \in \{1, \ldots, q\}$ and $j \in \{1, \ldots, q\}$ of Figure 7 is used to implement the effect of action $\alpha_j(\overline{x})$ on attribute variables f_i. Its definition is given in Figure 8. In particular, upd_i^j is defined by way of auxiliary function *enum*. For vector $\overline{z} = (z_1, \ldots, z_o)$, $y_k :: \overline{z}$ is equivalent notation for $(y_k, z_1, \ldots, z_o)$ and $[\,]$ stands for the empty vector. For each type T_k, we assume the existence of functions $first_k$ that returns the first element of type T, $last_k$ that returns the last element of type T_k, and $next_k\,(x)$ that returns the successor of x on condition that the type of variable x is T_k (following the total order induced by **ord**). For example, let $T_k = \{m_1, m_2, m_3\}$. Then, it is $first_k = m_1$, $next_k(m_1) = m_2$, $next_k(m_2) = m_3$ and $last_k = m_3$. According to the definition of "$enum\,(y_x :: \overline{z}, B)$" of Figure 8, the **loop** structure is employed to assign to y_k the current element of domain D_k (the first element of D_k is assigned at first) and a recursive call

to $enum\,(\overline{z}, B)$ is taken. If y_k has not been assigned the last value of D_k, which is expressed by condition $y_k \neq last_k$, the next value of D_k is assigned to y_k via $y_k := next_k\,(y_k)$ and $enum\,(\overline{z}, B)$ is called once again. Otherwise, the program breaks from the **loop** structure. This approach guarantees that, when computing the effect of actions on attribute variables via assignment B, all attribute variables are taken into consideration.

Note also that *enum* depends on function *mod* (E) which transforms an expression E by syntactically replacing function calls by array accesses as described previously. Namely according to the definition of $mod\,(w_i^j)$ of Figure 8, expressions of the form "$f_h\,(\mathsf{T},\ \overline{y})$" are replaced by $f_h\,[\mathbf{ord}\,(\overline{y})]$ and expressions of the form "$f_h\,(front\,(\mathsf{T}),\ \overline{y})$" are replaced by $g_h\,[\mathbf{ord}\,(\overline{y})]$ for all $h < n$. Recall at this point that the initial values of *attribute variables* $\overline{g}$ are by convection equal to $\perp$, which entails the replacement of *attribute function* calls of the form "$f_h\ (front\,(\mathsf{T}), \overline{y})$" in EB^3 type (1) value expressions w_i^0 by $\perp$ for all $h \in \{1, \ldots, n\}$. Notation $\overline{y} : \overline{T}_{at}$ is an abbreviation for "$y_1 : T_1, \ldots, y_s : T_s$" and $\overline{T}$ denotes the vector $(T, \ldots, T)$ of size n that is equal to the number of *attribute function* names in the system.

Expression $next_k\,(x)$ is implemented as LNT expression "**val** (**ord** $(x)+1$)". Note that in LNT assignment "$x :=$ **val** (i)" identifier **val** stands for the predefined LNT function that returns the i-th element of given ordered set denoting the domain of x's type. Such functions are available in LNT for all finite types.

Similarly, assuming common parameter vector $\overline{x} \doteq (x_1, \ldots, x_p)$ for all action labels α_j for all $j \in \{1, \ldots, q\}$ or, equivalently, action name definitions of the form $\alpha_j(x_1 : T_1, \ldots, x_p : T_p)$ for all $j \in \{1, \ldots, q\}$, we define "$\overline{x} : \overline{T}_{ac}$" that is equivalent to "$x_1 : T_1, \ldots, x_p : T_p$", where types $T_1, \ldots, T_p$ are not to be confused with the types corresponding to the *attribute* parameters $y_k \in \overline{y}$ for $k \in \{1, \ldots, s\}$ as seen earlier. Then, upon synchronization on action $\alpha_j\,(?\overline{x}, !\,\overline{f})$ with the LNT process corresponding to EB^3's *main* process (see translation of processes below), the values of all attribute variables f_i for $i \in \{1, \ldots, n\}$ are updated using function upd_i^j.

As an example, we demonstrate how the definition of Figure 7 regarding the memory process, applies to the EB^3 specification of the library management system for one book and two members. Member and book IDs are defined in the LNT program as follows:

type *MID* **is** $m_1, m_2, m_\perp$ **with** "**eq**","**ne**","**ord**","**val**" **end type**
type *BID* **is** $b_1, b_\perp$ **with** "**eq**","**ne**","**ord**","**val**" **end type**

Identifier **eq** (**ne**) denotes the equality (inequality) operator among member IDs or book IDs. The bottom value for member IDs is denoted as $m_\perp$ and the bottom value for book IDs is denoted as $b_\perp$. In the following, the type referring to the number of books possessed by each member of the library is denoted as an array *NB* and the type referring to the current borrower of each book is denoted as an array *BOR*. Hence, *NB* and *BOR* are defined in LNT as follows:

type *NB* **is array**[0..2] **of** *NAT* **end type**
type *BOR* **is array**[0..1] **of** *MID* **end type**

Now, we need to explain how the execution of communication label *LEND* modifies *attribute variable* vectors "*borrower* : *BOR*" and "*nbLoans* : *NB*". First, we remark that, according to the EB^3 specification of the library management system, the execution of EB^3 action "*Lend* (bId, mId)" modifies *attribute variables* $borrower[bId]$ and $nbLoans[mId]$. Hence, following the definition of Figure 7, we need an auxilliary variable bId' to go through all possible values of book IDs, i.e. the elements that inhabit *BID*'s domain, in order to simulate the modification of $borrower[bId]$ in case that book bId is lent to member mId. In each iteration of the **loop** construct, it is checked if bId is equal to the current value of bId', in which case $borrower[bId]$ is set to the current borrower, i.e. mId. Then, it is checked if bId' is equal to the last element of *BID*'s domain, i.e. b_1, in which case the program control breaks from the **loop** construct. Otherwise, bId' is assigned the next element in the ordered set of *BID*'s domain, i.e. "$bId' :=$ **val** (**ord** $(bId') + 1$)", and the execution of the **loop** construct is repeated.

Similarly, we need an auxiliary variable mId' to go through all possible values of member IDs, i.e. the elements that inhabit *MID*'s domain, in order to simulate the modification of $nbLoans[bId]$ in case that book bId is lent to member mId. In each iteration of the **loop** construct, it is checked if mId' coincides with the current value of mId', in which case $nbLoans[mId]$ is increased by one. Then, it is checked if mId' is equal to the last element of *MID*'s domain, i.e. m_2, in which case the program control breaks from the **loop** construct. Otherwise, mId' is assigned the next element in the ordered set of *MID*'s domain, i.e. "$mId' :=$ **val** (**ord** $(mId') + 1$)", and the execution of the **loop** construct is repeated.

The code for the memory M is then given as follows:

```
process M [Acquire, Discard, Register, Unregister, Lend, Return : any] is
  var mId : MID, bId : BID, bId' : BID,
      mId' : MID, borrower : BOR, nbLoans : NB  in
      borrower := BOR(m⊥);  nbLoans := NB(0);
  loop
    select
        Acquire (?bId)
    []  Discard (?bId, ?borrower)
    []  Register (?mId)
    []  Unregister (?mId)        []  Lend (?bId, ?mId, !nbLoans, !borrower);
        bId' := b1;
        loop L1 in
          if (bId' eq bId) then
            borrower [ord (bId')] := mId
          end if;
          if (bId' eq b1) then
            break L1
          else
            bId' := val (ord (bId') + 1)
          end if;
        end loop
        mId' := m1;
        loop L2 in
          if (mId' eq mId) then
            nbLoans [ord (mId')] := nbLoans [ord (mId')] + 1
          end if;
          if (mId' eq m2) then
            break L2
          else
            mId' := val (ord (mId') + 1)
          end if
        end loop
    []  RET (?bId);
        mId' := m1;
        loop L1 in
          if (mId' eq borrower [ord (bId)]) then
            nbLoans [ord (mId')] := nbLoans [ord (mId')] − 1
          end if;
          if (mId' eq m2) then
            break L1
          else
            mId' := val (ord (mId') + 1)
          end if
        end loop
```

```
            bId' := b1;
            loop L1 in
              if (bId' eq bId) then
                borrower [ord (bId)] := m⊥
              end if;
              if (bId' eq b1) then
                break L1
              else
                bId' := val (ord (bId') + 1)
              end if
            end loop;
        end select
      end loop
      end var
    end process
```

Note that LNT statement "*borrower* := $BOR(m_\perp)$" is equivalent to "*borrower*[**ord** (b_1)] := $m_\perp$". Similarly, "*nbLoans* := $NB(0)$" is equivalent to "*nbLoans*[**ord** (m_1)] := 0; *nbLoans*[**ord** (m_2)] := 0". Note that the initial number of loans is set to 0, whereas according to Figure 1 it should be set to $\perp$. The reason is that it is impossible to define $Nat_\perp$ of Figure 1 without recourse to complex LNT data structures. Hence, the symbol $\perp$ of Figure 1 is basically matched to the symbol 0 in the above LNT program, the symbol 0 of Figure 1 is matched to 1 in the above LNT program etc.

9.2. Optimizations.

Notice that the inert action λ has been removed from the previous LNT specification, since λ is does not appear in the program script and, as a result, it is supposed not to affect the control flow.

Similarly, *attribute variable* vector $\overline{f} = (nbLoans, borrower)$ is removed from the parameter vector of communication labels that synchronize with expressions of the form "$\alpha\,(\overline{v}, ?\overline{f})$ **where** $mod\,(C)$" present in other LNT processes (see Figure 9), for which the corresponding guard $mod\,(C)$ makes no use of *attribute variable* vector $\overline{f}$. In particular, this optimization applies to communication labels *Acquire*, *Register*, *Unregister* and *Return*. The static analysis of LNT specifications cater for an efficient discovery of all guards, as well as their related communication labels.

The only coordinates of *attribute variable* vector $\overline{f} = (nbLoans, borrower)$ passed as parameters to the communication labels "$\alpha\,(\overline{v}, ?\overline{f})$ **where** $mod\,(C)$" are exactly those appearing in $mod\,(C)$. Notice, for example, that vector *nbLoans* is not a subset of the parameter vector passed to communication label *Discard*, since the static analysis of the LNT specification reveals that no corresponding guard to *Discard* makes use of *nbLoans*.

The code referring to *attribute variables* that remain unchanged during a communication is omitted. This is the case for communication labels *Acquire*, *Discard*, *Register* and *Unregister* in the previous LNT code.

It turns out that the code simulating the effect of EB^3 action "*Lend* (bId, mId)" on *attribute variables* *borrower*[*bId*] and *nbLoans*[*mId*] can be optimized based on the simple observation that *attribute variables* *borrower*[*bId'*] for $bId' \neq bId$ and *nbLoans*[*mId'*] for $mId' \neq mId$ remain unaffected. Hence, the corresponding part of process M regarding EB^3 action "*Lend* (bId, mId)" can be modified as follows:

```
[]  Lend (?bId, ?mId, !nbLoans, !borrower);
      borrower [ord (bId)] := mId;
      nbLoans [ord (mId)] := nbLoans [ord (mId)] + 1
```

A similar approach applies to communication label *Ret*. The optimized LNT specification of the library management system is given in the appendix. Note that this optimization technique is applicable whenever the **in** offers of the communication label in question, i.e. the parameters marked with ?, suffice to determine

$$
\begin{array}{rcll}
t\,(\lambda, C) & = & \lambda\,(?\overline{f})\ \textbf{where}\ \mathit{mod}\,(C) & (1)\\
t\,(\alpha\,(\overline{v}), C) & = & \alpha\,(\overline{v}, ?\overline{f})\ \textbf{where}\ \mathit{mod}\,(C) & (2)\\
t\,(E_1.E_2, C) & = & t\,(E_1, C);\ t\,(E_2, \mathrm{true}) & (3)\\
t\,(C' \Rightarrow E_0, C) & = & t\,(E_0, C\ \textbf{andthen}\ C') & (4)\\
t\,(E_1 \mid E_2, C) & = & \textbf{select}\ t\,(E_1, C)\ \texttt{[]}\ t\,(E_2, C)\ \textbf{end select} & (5)\\
t\,(\mid x\colon V \colon E_0, C) & = & \textbf{var}\ x := \textbf{any}\ V;\ t\,(E_0, C)\ \textbf{end var} & (6)\\
t\,({E_0}^*, \mathrm{true}) & = & \textbf{loop}\ L_{E_0}\ \textbf{in} & \\
 & & \quad \textbf{select} & \\
 & & \quad\quad \lambda\,(?\overline{f});\ \textbf{break}\ L_{E_0}\ \texttt{[]}\ t\,(E_0, \mathrm{true}) & \\
 & & \quad \textbf{end select} & \\
 & & \textbf{end loop} & (7)\\
t\,(E_1\ \texttt{|[}\Delta\texttt{]|}\ E_2, \mathrm{true}) & = & \textbf{par}\ \Delta\ \textbf{in}\ t\,(E_1, \mathrm{true})\ \texttt{||}\ t\,(E_2, \mathrm{true})\ \textbf{end par} & (8)\\
t\,(\texttt{|[}\Delta\texttt{]|}\, x\colon V \colon E_0, \mathrm{true}) & = & \textbf{par}\ \Delta\ \textbf{in}\ E_0[x := v_1]\ \texttt{||}\ \ldots\ \texttt{||}\ E_0[x := v_n]\ \textbf{end par} & \\
 & & \quad \text{where } V = \{v_1, \ldots, v_n\} & (9)\\
t\,(P\,(\overline{v}), \mathrm{true}) & = & P\ \texttt{[}\alpha_1, \ldots, \alpha_q, \lambda\texttt{]}\ (\overline{v}) & (10)
\end{array}
$$

In all other cases:

$$
t\,(E_0, C) = \begin{cases}
\textbf{if}\ \mathit{mod}\,(C)\ \textbf{then}\ t\,(E_0, \mathrm{true})\ \textbf{else stop end if} & \\
\quad \text{if } C \text{ does not use attribute functions} & \\
\textbf{par}\ \alpha_1, \ldots, \alpha_q, \lambda\ \textbf{in} & \\
\quad t\,(E_0, \mathrm{true}) & \\
\texttt{||}\ pr_C\ \texttt{[}\alpha_1, \ldots, \alpha_q, \lambda\texttt{]}\ (\mathit{vars}\,(C)) & \\
\textbf{end par} \qquad \text{otherwise} &
\end{cases} \quad (11)
$$

Fig. 9. Translation from EB^3 process to LNT process

which *attribute variables* are modified. Based on our experience with EB^3 specifications, this optimization technique is often applicable on the corresponding LNT specifications. It is also incorporated on our tool EB^{3}2LNT.

9.3. Translation of processes.

We define a translation function t from an EB^3 process expression to an LNT process. Most EB^3 constructs are process algebra constructs with a direct correspondence in LNT. The main difficulty arises in the translation of guarded process expressions of the form "$C \Rightarrow E_0$" in a way that guarantees the *guard-action atomicity*. This led us to consider a second parameter for the translation function t, namely the condition C, whose evaluation is delayed until the first action occurring in the process expression E_0. The definition of $t\,(E, C)$ is given in Figure 9. An EB^3 specification E_0 will then be translated into:

par $\alpha_1, \ldots, \alpha_q, \lambda$ **in** $t\,(E_0, \mathrm{true})$ || M [$\alpha_1, \ldots, \alpha_q, \lambda$] **end par**

and every process definition of the form "$P\,(\overline{x}) = E$" will be translated into the process:

process P [$\alpha_1, \ldots, \alpha_q, \lambda$: **any**] ($\overline{x}$: type ($\overline{x}$)) **is** $t\,(E, \mathrm{true})$ **end process**,

where $\{\alpha_1, \ldots, \alpha_q\} = \mathit{Lab}$. The Rules of Figure 9 can be commented as follows:

- Rule (1) translates the λ action. Note that λ cannot be translated to the empty LNT statement **null**, because execution of λ may depend on a guard C, whose evaluation requires the memory to be read,

so as to get *attribute variable* values. This is done by the LNT communication action $\lambda\,(?\overline{f})$. The guard C is evaluated after replacing calls to attribute functions (all of which have the form $f_i\,(\mathsf{T}, \overline{v_i})$) by the appropriate *attribute variables*, using function *mod* defined in Figure 7. Rule (2) is similar but handles visible actions.

- Rule (3) translates EB^3 sequential composition into LNT sequential composition, passing the evaluation of C to the first process expression.

- Rule (4) makes a conjunction between the guard of the current process expression with the guard already accumulated from the context.

- Rules (5) and (6) translate the choice and quantified choice operators of EB^3 into their direct LNT counterpart.

- Rule (7) translates the Kleene closure into a combination of LNT loop and select, following the identity ${E_0}^* = \lambda \mid E_0.{E_0}^*$.

- Rule (8) translates EB^3 parallel composition into LNT parallel composition.

- Rule (9) translates EB^3 quantified parallel composition into LNT parallel composition by expanding the type V of the quantification variable, since LNT does not have a quantified parallel composition operator.

- Rule (10) translates an EB^3 process call into the corresponding LNT process call, which requires gates to be passed as parameters.

- Rules (7) to (10) only apply when the guard C is trivially true. In the other cases, we must apply Rule (11), which generates code implementing the guard. If C does not use attribute functions, i.e., does not depend on the trace, then it can be evaluated immediately without communicating with the memory process (first case). Otherwise, the guard evaluation must be delayed until the first action of the process expression E_0. When E_0 is either a Kleene closure, a parallel composition, or a process call, identifying its first action syntactically is not obvious. One solution would consist in expanding E_0 into a choice in which every branch has a fixed initial action[3], to which the guard would be added. We preferred an alternative solution that avoids the potential combinatorial explosion of code due to static expansion. A process pr_C (defined in Figure 10) is placed in parallel to $t\,(E_0, \mathrm{true})$ and both processes synchronize on all actions. Process pr_C imposes on $t\,(E_0, \mathrm{true})$ the constraint that the first executed action must satisfy the condition C (**then** branch). For subsequent actions, the condition is relaxed (**else** branch).

[3] Such a form, commonly called *head normal form* [BPS01], is used principally in the context of the process algebra ACP [BK85] to analyse the behaviour of recursive processes.

```
process pr_C [α_1, ..., α_q, λ : any] (vars (C) : type (vars (C))) is
var start : bool,   x̄ : T̄_ac,   f̄ : T̄ in
  start := true;
  loop L in select
    if start then
      start := false;
      select
        α_1 (?x̄, ?f̄) where mod (C)
        [] ... []
        α_q (?x̄, ?f̄) where mod (C)
        []
        λ (?f̄) where mod (C)
      end select
    else
      select
        α_1 (?x̄, ?f̄)
        [] ... []
        α_q (?x̄, ?f̄)
        []
        λ (?f̄)
      end select
    end if
  [] break L end select end loop
end var
end process
```

Fig. 10. Process pr_C

Example Revisited. The **main** process of the LNT specification describing the library management system is defined as follows:

```
process Main [Acquire, Discard, Register, Unregister, Lend, Return : any] is
  par Acquire, Discard, Register, Unregister, Lend, Return in
    par
      loop L in
        select break L [] book [Acquire, Discard] (b_1)
        end select
      end loop
    ||
      par
        loop L in
          select break L [] member [Register, Unregister, Lend, Return] (m_1)
          end select
        end loop
      ||
        loop L in
          select break L [] member [Register, Unregister, Lend, Return] (m_2)
          end select
        end loop
      end par
    end par
```

```
    ||
      M [Acquire, Discard, Register, Unregister, Lend, Return]
    end par
  end process
```

Note that the equivant LNT process of EB^3 process *main* is placed in parallel to memory process M. Moreover, the simulation of EB^3 quantified parallel synchronization operator (that has no equivalent operator in LNT) is applied to EB^3 expressions "$|||\, bId : BID : book\,(bId)^*$" and "$|||\, mId : MID : member\,(mId)^*$" and the EB^3 Kleene Closure operator is simulated as described in Figure 9.
The body of LNT processes *book* and *member* is defined as below:

```
  process book [Acquire, Discard : any] (bId : BOOKID) is
    var borrower : BOR in
      Acquire(bid); Discard (bId, ?borrower) where (borrower [ord (bId)] eq m⊥)
    end var
  end process

  process member [Register, Unregister, Lend, Return : any] (mId : MEMBERID) is
    Register (mId);
    loop L in
      select break L [] loan [Lend, Return] (mId, b1)
      end select
    end loop;
    Unregister (mId)
  end process
```

The definition of LNT process *member* depends on LNT process *loan*, whose definition is given below:

```
  process loan [Lend, Return : any] (mId : MEMBERID, bId : BOOKID) is
    var borrower : BOR, nbLoans : NB in (* NbLoans is set to 1 *) in
      Lend (bId, mId, ?nbLoans, ?borrower)
        where ((borrower [ord (bId)] eq m⊥) and (nbLoans [ord (mId)] eq 1));
      Return (bId)
    end var
  end process
```

The following example illustrates and justifies the use of process pr_C as a means to solve the *guard-action atomicity* problem. Consider the EB^3 system:

$$C \Rightarrow Lend\,(b_1,\ m_1)\ |||\ Return\,(b_2),$$

where C denotes the Boolean condition:

$$borrower\,(\mathsf{T},\ b_1) = \bot\ \wedge\ nbLoans\,(\mathsf{T},\ m_1) < NbLoans$$

and $Lab = \{Lend, Return\}$. The LNT code corresponding to this system is the following:

```
  par Lend, Return, λ in
    par Lend, Return, λ in
      par Lend (b1, m1, ?f) || Return (b2, ?f) end par
    || pr_C [Lend, Return, λ] (b1, m1)
    end par
  || M [Lend, Return, λ]
  end par
```

The first action executed by this system may be either *Lend* or *Return*. We consider the case where

Lend is executed first. According to the LNT semantics, it results from the multiway synchronization of the following three actions:

- "*Lend* $(b_1,\ m_1, ?\overline{f})$" in the above expression,
- "*Lend* $(?b, ?m, ?\overline{f})$ **where** $borrower[\mathbf{ord}(b_1)] = \bot\ \wedge\ nbLoans[\mathbf{ord}(m_1)] < NbLoans$" in process pr_C (at this moment, *start* is true, see Fig. 10), and
- "*Lend* $(?b, ?m, !\overline{f})$" in process M (see Fig. 7).

Thus, in pr_C at synchronization time, $\overline{f}$ is an up-to-date copy of the memory stored in M, $b = b_1$, and $m = m_1$. The only condition for the synchronization to occur is the guard $mod\,(C)$, whose value is computed using the up-to-date copy $\overline{f}$ of the memory. In case $mod\,(C)$ evaluates to true, no other action (susceptible to modifying $\overline{f}$) can occur between the evaluation of $mod\,(C)$ and the occurrence of *Lend* as both happen synchronously, thus achieving the *guard-action atomicity*. Once *Lend* has occurred, *Return* can occur without any condition, as the value of *start* has now become false.

We developed an automatic translator tool from EB^3 specifications to LNT, named EB^{3}2LNT, implemented using the Ocaml Lex/Yacc compiler construction technology. It consists of about 900 lines of OCaml code. We applied EB^{3}2LNT on a benchmark of EB^3 specifications, which includes variations of the library management system.

We noticed that, for each EB^3 specification, the code size of the equivalent LNT specification is twice as big. Part of this expansion is caused by the fact that LNT is more structured than EB^3: LNT requires more keywords and gates have to be declared and passed as parameters to each process call. By looking at the Rules of Figure 9, we can see that the other causes of expansion are Rule (5), which duplicates the condition C, and Rule (9), which duplicates the body E_0 of the quantified parallel composition operator "$|[\Delta]|\, x : V : E_0$" as many times as there are elements in the set V. Both expansions are linear in the size of the source EB^3 code. However, in the case of a nested parallel composition "$|[\Delta_1]|\, x_1 : V_1 : \ldots\ |[\Delta_n]|\, x_n : V_n : E_0$", the expansion factor is as high as the product of the number of elements in the respective sets $V_1, \ldots, V_n$, which may be large. If E_0 is a big process expression, the expansion can be limited by encapsulating E_0 in a parameterized process "$P_{E_0}\,(x_1, \ldots, x_n)$" and replacing duplicated occurrences of E_0 by appropriate instances of P_{E_0}.

10. Case Study Revisited

We illustrate below the application of the EB^{3}2LNT translator in conjunction with CADP for analyzing an extended version of the IS library management system, whose description in EB^3 can be found in Annex C of [Ger06]. With respect to the simplified version presented in Section 3, the IS enables e.g., members to renew their loans and to reserve books, and their reservations to be cancelled or transferred to other members on demand. The desired behaviour of this IS was characterized in [FFC10] as a set of 15 requirements expressed informally as follows:

R1. A book can always be acquired by the library when it is not currently acquired.
R2. A book cannot be acquired by the library if it is already acquired.
R3. An acquired book can be discarded only if it is neither borrowed nor reserved.
R4. A person must be a member of the library in order to borrow a book.
R5. A book can be reserved only if it has been borrowed or already reserved by some member.
R6. A book cannot be reserved by the member who is borrowing it.
R7. A book cannot be reserved by a member who is reserving it.
R8. A book cannot be lent to a member if it is reserved.
R9. A member cannot renew a loan or give the book to another member if the book is reserved.
R10. A member is allowed to take a reserved book only if he owns the oldest reservation.
R11. A book can be taken only if it is not borrowed.
R12. A member who has reserved a book can cancel the reservation at anytime before he takes it.
R13. A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or cancelled.
R14. Ultimately, there is always a procedure that enables a member to leave the library.

Table 1. Model checking results for the library management system

(m, p)	(3,2)	(3,3)	(3,4)	(4,3)
states	1,002	182,266	8,269,754	27,204,016
trans.	5,732	1,782,348	105,481,364	330,988,232
time	1.9s	14.4s	31'39s	140'22s
R1	0.3s	1.8s	5'19s	20'13s
R2	0.2s	2.9s	9'26s	36'7s
R3	0.2s	9.4s	97'46s	26'47s
R4	0.2s	1.7s	5'15s	18'40s
R5	0.2s	2.2s	6'46s	21'52s
R6	0.2s	4.1s	38'30s	10'19s
R7	0.2s	7.4s	65'22s	24'33s
R8	0.2s	2.2s	6'52s	22'27s
R9	0.2s	2.3s	6'38s	22'29s
R10	0.3s	13.3s	43'59s	62'07s
R11	0.3s	2.5s	6'36s	22'14s
R12	0.3s	4.0s	10'47s	45'09s
R13	0.4s	4.3s	11'46s	36'07s
R14	0.3s	3.6s	10'41s	37'33s
R15	0.2s	2.8s	7'53s	28'56s

R15. A member cannot borrow more than the loan limit defined at the system level for all users.

We expressed all the above requirements using the property specification language MCL [MT08]. We show below the MCL formulation of two requirements from the list above, which denote typical safety and liveness properties. Requirement R2 is expressed in MCL as follows:

$$[\,\mathbf{true}^*.\{\texttt{ACQUIRE ?}B : \mathbf{string}\}.(\mathbf{not}\ \{\texttt{DISCARD !}B\})^*.\{\texttt{ACQUIRE !}B\}\,]\ \mathbf{false}$$

This formula uses the standard *safety* pattern "$[\beta]$ **false**", which forbids the existence of transition sequences matching the regular formula β. Here the undesirable sequences are those containing two *Acquire* operations for the same book B without a *Discard* operation for B in the meantime. The regular formula $\mathbf{true}^*$ matches a subsequence of (zero or more) transitions labeled by arbitrary actions. Note the use of the construct "$\texttt{?}B : \mathbf{string}$", which matches any string and extracts its value in the variable B used later in the formula. Therefore, the above formula captures all occurrences of books carried by *Acquire* operations in the model. Requirement R12 is formulated in MCL as follows:

$$\begin{array}{l}[\,\mathbf{true}^*.\{\texttt{RESERVE ?}M : \mathbf{string}\ \texttt{?}B : \mathbf{string}\}.\\ \ (\mathbf{not}\ (\{\texttt{TAKE !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{TRANSFER !}M\ \texttt{!}B\}))^*\,]\\ \quad \langle\,(\mathbf{not}\ (\{\texttt{TAKE !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{TRANSFER !}M\ \texttt{!}B\}))^*.\ \{\texttt{CANCEL !}M\ \texttt{!}B\}\,\rangle\ \mathbf{true}\end{array}$$

This formula denotes a *liveness* property of the form "$[\beta_1]\ \langle\beta_2\rangle$ **true**", which states that every transition sequence matching the regular formula β_1 (in this case, book B has been reserved by member M and subsequently neither taken nor transferred) ends in a state from which there exists a transition sequence matching the regular formula β_2 (in this case, the reservation can be cancelled before being taken or transferred).

Requirement R7 can be formulated as follows:

$$\begin{array}{l}[\,\mathbf{true}^*\,.\,\{\texttt{RESERVE ?}M : \mathbf{string}\ \texttt{?}B : \mathbf{string}\}\,.\,(\mathbf{not}\ (\{\texttt{TAKE !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{CANCEL !}M\ \texttt{!}B\}))^*\,.\\ \quad \{\texttt{RESERVE !}M\ \texttt{!}B\}\,]\ \mathbf{false}\end{array}$$

All requirements formulated in MCL can be found in Appendix C.

Using EB^{3}2LNT, we translated the EB^3 specification of the library management system to LNT. The resulting specification was checked against all the 15 requirements, formulated in MCL, using the EVALUATOR 4.0 model checker of CADP. The experiments were performed on an Intel(R) Core(TM) i7 CPU 880 at 3.07GHz. Table 1 shows the results for several configurations of the IS, obtained by instantiating the number of books (m) and members (p) in the IS. All requirements were shown to be valid on the IS specification. The second and third line of the table indicate the number of states and transitions of the LTS corresponding to the LNT specification. The fourth line gives the time needed to generate the LTS and the other lines give the verification time for each requirement.

Note that the number of states generated increases with the size of m and p as EVALUATOR 4.0 applies explicit techniques for state space generation. According to Table 1, verifying the library management system becomes intractable for more than four members and four books. This misbehaviour is mainly attributed to the presence of parallel compositions between members and books as well as the loose synchronisation between these components leading to exponential growth of the state space.

Comparison with [FFC10] We recall that CADP was used in [FFC10] for the verification of a version of the library management system with similar functionalities. According to the same paper, the time needed to generate the corresponding LTS for $NbLoans = 1$, was 970.2 sec approximately, the average time needed to verify the system requirements was 74.63 sec and no information is provided for the size of the LTS, which means that no reliable comparison of the two works is possible. Note also that the LNT specification, to whom the results of Table 1 are related, was automatically generated by EB^{3}2LNT, whereas the results of [FFC10] correspond to a manual LNT specification of the library management system.

Regarding the other verification tools employed in [FFC10], only verification with ALLOY outperforms our verification scheme with CADP, which is normal as in [FFC10] in the case of reachability properties for example the traces needed to satisfy them are specified by the user, which implies that automatic verification with ALLOY as done in [FFC10] is not automatic and therefore not adequate.

11. Conclusion

We proposed an approach for equipping the EB^3 method with formal verification capabilities by reusing already available model checking technology. Our approach relies upon a new translation from EB^3 to LNT, which provides a direct connection to all the state-of-the-art verification features of the CADP toolbox. The translation, based on alternative memory semantics of EB^3 [VD13] instead of the original trace semantics [FSt03], was automated by the EB^{3}2LNT translator and validated on several examples of typical ISs. So far, we experimented only the model checking of MCL data-based temporal properties on EB^3 specifications. However, CADP also provides extensive support for equivalence checking and compositional LTS construction, which can be of interest to IS designers. We also provided a formal proof of the translation from EB^3 to LNT that can be found in [Vek14], which could serve as reference for translating EB^3 to other process algebras as well.

EB^{3}2LNT offers insight to interested readers on:

- how to code global state as a process running parallel to the principal system specification, and
- how to solve the atomicity problem (imposing the simultaneous evaluation of C with the execution of the first action of E in expressions of the form $C \Rightarrow E$) by a way of another process running to the principal system specification.

On the other hand, the main limitation of the proposed translation is that model-checking becomes intractable for specifications describing multiple concurrent processes as is the case for the library specification for more than four members and four books. This inconvenience is attributed mainly to:

- the frequent use of parallel composition operators in EB^{3}specifications that lead to the exponential growth of execution traces, and
- the fact that CADP employs explicit techniques instead of symbolic techniques for state space generation.

Another drawback of the proposed translation arises at the stage of property verification. Properties should be expressed in MCL, which implies that EB^3 users should be familiar with MCL's syntax. However, one should note that the whole scale of MCL formulas can be used for the verification of EB^{3}specifications, despite the fact that the emphasis of this paper is given on *safety* and *liveness* properties.

As future work, we plan to study abstraction techniques for verifying properties regardless of the number of entity instances that participate in the IS, following the approaches for parameterized model checking [ABJ$^+$99]. This study will establish the fact that verifying systems with many components (members and books in our case) is equivalent to verifying systems with the same structure as before, but less components. In particular, we will observe how the insertion of new functionalities into an IS affects this issue, and we will formalize this in the context of EB^3 specifications.

References

[ABJ+99] P.A. Abdulla, A. Bouajjani, B. Jonsson, M. Nilsson. Handling Global Conditions in Parameterized System Verification. In *Proceedings of CAV*, LNCS vol. 1633, pages 134–145, Springer, 1999.

[Abr05] J.R. Abrial. *The B-Book - Assigning programs to meanings.* Cambridge University Press, 2005.

[Abr10] J.R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[BB02] H.R. Barradas, D. Bert. Specification and Proof of Liveness Properties under Fairness Assumptions in B Event Systems. In *Proceedings of Integrated Formal Methods*, LNCS vol. 2335, pages 360–379, Springer, 2002.

[BCC+99] A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS vol. 1579, pages 193–207, Springer, 1999.

[BCJ02] F. Bellegarde, S. Chouali, J. Julliand. Verification of Dynamic Constraints for B Event Systems under Fairness Assumptions. In *ZB 2002:Formal Specification and Development in Z and B*, LNCS vol. 2272, pages 477–496, Springer, 2002.

[BPS01] J.A. Bergstra, A. Ponse, S.A. Smolka. *Handbook of Process Algebra.* Elsevier, 2001.

[BK85] J.A. Bergstra, J.W. Klop. Algebra of Communicating Processes with Abstraction. *TCS*, 37:77–121, 1985.

[Cho10] R. Chossart. Évaluation d'outils de vérification pour les spécifications de systèmes d'information. Master's thesis, Université de Sherbrooke, 2010.

[Cl] ClearSy. *Atelier B.* http://www.atelierb.societe.com.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. Springer, 2002.

[CCG05] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, G. Smeding. *Reference Manual of the LOTOS NT to LOTOS Translator – Version 5.4.* INRIA/VASY, 2011.

[CES86] E. M. Clarke, E. A. Emerson, A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications In *Journal of ACM Transactions on Programming Languages and Systems*, vol. 8, pages 244–263, Springer, 1986.

[EL+86] E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of Logic in Computer Science*, pages 267–278, 1986.

[ETL+04] N. Evans, H. Treharne, R. Laleau, M. Frappier. How to verify dynamic properties of information systems. In *Workshop of Software Engineering and Formal Methods*, pages 416–425, 2004.

[FFC10] M. Frappier, B. Fraikin, R.Chossart, R. Chane-Yack-Fa, M. Ouenzar. Comparison of model checking tools for information systems. In *Proceedings of ICFEM*, LNCS vol. 6447, pages 581–596, Springer, 2010.

[Fdr97] Formal Systems (Europe) Ltd. Failures-Divergences Refinement. FDR2 User Manual 1997.

[FSt03] M. Frappier, R. St.- Denis. EB^3: an entity-based black-box specification method for information systems. In *Journal of Software and System Modeling*, vol. 2, pages 134–149, Springer, 2003.

[GLMS10] H. Garavel, F. Lang, R. Mateescu, W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, LNCS vol. 6605, pages 372–387, Springer, 2011.

[Ger06] F. Gervais. *Combinaison de spécifications formelles pour la modélisation des systèmes d'information.* PhD thesis, Université de Sherbrooke, 2006.

[Gro06] J. Groslambert. Verification of LTL on B Event System. Technical report, 2006.

[GFL05] F. Gervais, M. Frappier, R. Laleau. Synthesizing B Specifications from EB^3 Attribute Definitions. In *Proceedings of Integrated Formal Methods*, LNCS vol. 3771, pages 207–226 Springer, 2005.

[GFL06] F. Gervais, M. Frappier, R. Laleau. Refinement of EB^3 Process Patterns into B Specifications. In *Proceedings of Formal Specification and Development in B*, LNCS vol. 4355, pages 201–215, Springer, 2006.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, vol. 21, pages 666–677, 1978.

[HA11] T-S. Hoang, J-R Abrial. Reasoning about liveness properties in Event-B. In *Proceedings of Formal Engineering Methods*, LNCS vol. 6991, pages 456–471, 2011.

[Hol04] G. J. Holzmann. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley 2004.

[Jac06] D. Jackson. Software Abstractions. MIT Press, 2006.

[JFG+10] M. E. Jiague, M. Frappier, F. Gervais, P. Konopacki, R. Laleau, J. Milhau, R. St-Denis. Model-Driven Engineering of Functional Security Policies. In *Proceedings of International Conference on Enterprise Information*, pages 374–379, 2010.

[Lot01] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard number 15437:2001, International Organization for Standardization — Information Technology, Genève, 2001.

[LB03] M. Leuschel, M. Butler. ProB: A model checker for B. In *Proceedings of Symposium on Formal Methods*, LNCS vol. 2805, pages 855–874, Springer-Verlag, 2003.

[Koz83] D. Kozen. Results on the Propositional μ-calculus. In *TCS*, vol. 27, 333–354, 1983.

[LMC00] M. Leuschel, T. Massart, A. Currie.: How to make FDR spin: LTL model checking of CSP by refinement. Technical report, 2000.

[Mor98] C .C. Morgan. Programming from Specifications, Prentice Hall, 1998.

[MF15] A. Mammar, M. Frappier. Proof-based verification approaches for dynamic properties: application to the information system domain. In *Journal of Formal Aspects of Computing*, vol. 27, pages 335–374, 2015.

[MIL+11] J. Milhau, A. Idani, R. Laleau, M.A. Labiadh, Y. Ledru, M. Frappier. Combining UML, ASTD and B for the formal specification of an access control filter. In *Journal of Innovations in Systems and Software Engineering*, vol. 7, pages 303–313, Springer, 2011.

[MT08] R. Mateescu, D. Thivolle. A model checking language for concurrent value-passing systems. In *Proceedings of Formal Methods*, LNCS vol. 5014, pages 148–164, Springer, 2008.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Journal of Foundations of Computer Science*, vol. 18, pages 46–57, Springer, 1977.

[QS83] J-P. Queille, J. Sifakis. Fairness and Related Properties in Transition Systems-A Temporal Logic to Deal with Fairness. In *Journal of Acta Informatica*, vol. 19, pages 195–220, 1983.

[Str82] R. Streett. Propositional Dynamic Logic of Looping and Converse. In *Information and Control*, vol. 54, pages 121–141, 1982.

[ST05] S. Schneider, H. Treharne. CSP Theorems for Communicating B Machines. In *Journal of Formal Aspects of Computing*, vol. 17, pages 390–422, Springer, 2005.

[STW+14] S. Schneider, H. Treharne, H. Wehrheim, D. M. Williams. Managing LTL Properties in Event-B Refinement. In *Proceedings of Integrated Formal Methods*, pages 221–237, Springer, 2014.

[TSB03] H. Treharne, S. Schneider, M. Bramble. Composing Specifications using Communication. In *Proceedings of ZB*, LNCS vol. 2651, pages 55–78, Springer, 2003.

[Vek14] D. Vekris. *Verification of* EB^3 *specifications with the aid of model-checking techniques.* https://tel.archives-ouvertes.fr/tel-01140261/document. PhD thesis, Université de Paris-Créteil, 2014.

[VD13] D. Vekris, C. Dima. Efficient Operational Semantics for EB^3 for Verification of Temporal Properties. In *Proceedings of Fundamentals of Software Engineering*, LNCS vol. 8161, pages 133–149, Springer, 2013.

[VLD+13] D. Vekris, F. Lang, C. Dima, R. Mateescu. Verification of EB^3 Specifications using CADP. In *Proceedings of Integrated Formal Methods*, LNCS vol. 7940, pages 61–76, Springer, 2013.

A. LNT Code for the Simplified Library Management System

We give the optimized LNT code for the simplified Library Management System, with 2 members, 1 book, and *NbLoans* set to 1.

```
module library is

type mId is m1, m2, m⊥ with "eq","ne","ord","val" end type
type BID is b1, b⊥ with "eq","ne","ord","val" end type
type NB is array[0..2] of NAT end type
type BOR is array[0..1] of MID end type

process M [Acquire, Discard, Register, Unregister, Lend, Return : any] is
  var mId : MID, bId : BID, bId' : BID,
      mId' : MID, borrower : BOR, nbLoans : NB in
      mId := m⊥;   borrower := BOR(m⊥);   nbLoans := NB(0);
  loop
    select
          Acquire (?bId)
    []    Discard (?bId, ?borrower)
    []    Register (?mId)
    []    Unregister (?mId)
    []    Lend (?bId, ?mId, !nbLoans, !borrower);
          borrower [ord (bId)] := mId;
          nbLoans [ord (mId)] := nbLoans [ord (mId)] + 1
    []    RET (?bId);
          mId' := borrower [ord (bId)];
          nbLoans [ord (mId')] := nbLoans [ord (mId')] − 1;
          borrower [ord (bId)] := m⊥
    end select
  end loop
  end var
end process

process loan [Lend, Return : any] (mId : mId, bId : BID) is
  var borrower : BOR, nbLoans : NB in (* NbLoans is set to 1 *) in
    Lend (bId, mId, ?nbLoans, ?borrower)
      where ((borrower [ord (bId)] eq m⊥) and (nbLoans [ord (mId)] eq 1));
    Return (bId)
  end var
end process

process book [Acquire, Discard : any] (bId : BID) is
  var borrower : BOR in
    Acquire(bId); Discard (bId, ?borrower) where (borrower [ord (bId)] eq m⊥)
  end var
end process

process member [Register, Unregister, Lend, Return : any] (mId : mId) is
  Register (mId);
  loop L in
    select break L [] loan [Lend, Return] (mId, b1)
    end select
  end loop;
  Unregister (mId)
end process
```

```
process Main [Acquire, Discard, Register, Unregister, Lend, Return : any] is
  par Acquire, Discard, Register, Unregister, Lend, Return in
    par
      loop L in
        select break L [] book [Acquire, Discard] (b1)
        end select
      end loop
    ||
      par
        loop L in
          select break L [] member [Register, Unregister, Lend, Return] (m1)
          end select
        end loop
      ||
        loop L in
          select break L [] member [Register, Unregister, Lend, Return] (m2)
          end select
        end loop
      end par
    end par
  ||
    M [Acquire, Discard, Register, Unregister, Lend, Return]
  end par
end process

end module
```

B. LNT Code for the Extended Library Management System

We give the LNT code for the extended Library Management System ($NbLoans = 1$). This code is generated by EB^{3}2LNT except the functions "*add_reservation*","*cancel_reservation*", "*last_reservation*", "*nil_reservation*" and "*is_reserved*", which are coded by hand, since our compiler does not support user-defined EB^3 types and (non-attribute) functions. We also define type "*RESERV*" to simplify the coding. By convention, for every type T we have $ord(\bot) = 0$, $ord(first_T) = 1$, etc.

```
module library is

type mId is m1, m2, m⊥ with "eq", "ne", "ord", "val" end type
type BID is b1, b2, b⊥ with "eq", "ne", "ord", "val" end type
type MIDLIST is
  NIL, CONS(HD : MID, TL : MIDLIST) with "eq","ne" end type
type ACQUIR is array[0..1] of BOOL end type
type BOR is array[0..1] of MID end type
type NB is array[0..1] of NAT end type
type RESERV is array[0..1] of MIDLIST end type

function add_reservation (m : MID, l : MIDLIST) : MIDLIST is
  case l in var temp_mem : MID, temp_list : MIDLIST in
      NIL → return CONS(m, NIL)
    | CONS(temp_mem, temp_list) →
        return CONS(temp_mem, add_reservation(m, temp_list))
  end case
end function

function cancel_reservation (m : MID, l : MIDLIST) : MIDLIST is
  case l in var temp_mem : MID, temp_list : MIDLIST in
      NIL → return NIL
    | CONS(temp_mem, temp_list) where (temp_mem eq m) → return temp_list
    | CONS(temp_mem, temp_list) →
        return CONS(temp_mem, cancel_reservation(m, temp_list))
  end case
end function

function nil_reservation (l : MIDLIST) : BOOL is
  case l in var temp_list : MIDLIST in
      NIL → return true
    | ANY MIDLIST → return false
  end case
end function

function is_reserved (m : MID, l : MIDLIST) : MIDLIST is
  case l in var temp_mem : MID, temp_list : MIDLIST in
      NIL → return false
    | CONS(temp_mem, temp_list) where (temp_mem eq m) → return true
    | CONS(temp_mem, temp_list) → return is_reserved(m, temp_list)
  end case
end function

function last_reservation (m : MID, l : MIDLIST) : BOOL is
  case l in var temp_mem : MID, temp_list : MIDLIST in
      NIL → return false
    | CONS(temp_mem, temp_list) where (temp_mem eq m) → return true
    | CONS(temp_mem, temp_list) → return false
  end case
end function
```

```
process M [Acquire, Discard, Register, Lend, Take, Renew, Return, Reserve,
           Cancel, Unregister, Transfer : any] is
  var mId : mId, bId : BID, acquired : ACQUIR,
      borrower : BOR, nbLoans : NB, reservation : RESERV in
      acquired := ACQUIR(false);   borrower := BOR(m⊥);
      nbLoans := NB(0);   reservation : RESERV(NIL);
  loop
    select
        Acquire (?bId);
        acquired [ord (bId)] := true
    []  Discard (?bId, !borrower, !reservation);
        acquired [ord ()] := false
    []  Register (?mId);
        nbLoans [ord (mId)] := 1
    []  Unregister (?mId !nbLoans !reservation)
        nbLoans [ord (mId)] := 0
    []  Lend (?bId, ?mId, !acquired, !borrower, !reservation, !nbloans);
        borrower [ord (bId)] := mId;
        nbLoans [ord (mId)] := nbLoans [ord (mId)] + 1
    []  Reserve (?mId, ?bId, !acquired, !borrower, !reservation);
        reservation [ord (bId)] := add_reservation(mId, reservation [ord (bId)])
    []  Take (?mId, ?bId, !acquired, !borrower, !reservation, !nbLoans);
        borrower [ord (bId)] := mId;
        nbLoans [ord (mId)] := nbLoans [ord (mId)] + 1;
        reservation [ord (bId)] := cancel_reservation(mId, reservation [ord (bId)])
    []  Cancel (?mId, ?bId);
        reservation [ord (bId)] := cancel_reservation(mId, reservation [ord (bId)])
    []  Transfer (?mId, ?bId, !acquired, !borrower, !reservation, !nbLoans);
        nbLoans [ord (borrower [ord (bId)])] := nbLoans [ord (borrower [ord (bId)])] − 1;
        borrower [ord (bId)] := mId;
        nbLoans [ord (mId)] := nbLoans [ord (mId)] + 1
    []  Renew (?bId, !acquired, !borrower, !reservation)
    []  Return (?bId, !acquired, !borrower);
        nbLoans [ord (borrower [ord (bId)])] := nbLoans [ord (borrower [ord (bId)])] − 1;
        borrower [ord (bId)] := m⊥
  end loop
  end var
end process

process book [Acquire, Discard : any] (bId : BID) is
  var acquired : ACQUIR, borrower : BOR, reservation : RESERV in
    Acquire(bId); Discard (bId, ?borrower, ?reservation)
      where ((borrower [ord (bId)] eq m⊥) and
             (nil_reservation (reservation[ord (bId)]) eq true))
  end var
end process
```

```
process member [Lend, Take, Renew, Return, Reserve, Cancel, Transfer : any]
                (mId : MID, bId : BID) is
  var acquired : ACQUIR, borrower : BOR, reservation : RESERV, nbLoans : NB in
    select
      Lend (!bId, !mId, ?acquired, ?borrower, ?reservation, ?nbloans)
        where ((acquired [ord (bId)] eq true) and (borrower[ord (bId)] eq m⊥) and
               (nbLoans[ord (mId)] lt 2) and
               (nil_reservation (reservation [ord (bId)] eq true))
      loop L in select break L []
          Renew (!bId, ?acquired, ?borrower, ?reservation)
            where ((acquired [ord (bId)] eq true) and (borrower[ord (bId)] eq mId) and
                   (nil_reservation (reservation [ord (bId)]) eq true))
      end select end loop;
      select
          Return (!bId, ?acquired, ?borrower)
            where ((acquired [ord (bId)] eq true) and (borrower[ord (bId)] eq mId))
      [] var mId' : MID in
          mId' := any MID where ((mId' ne m⊥) and (mId' ne m⊥));
          Transfer (!mId', !bId, ?acquired, ?borrower, ?reservation, ?nbLoans)
            where ((acquired [ord (bId)] eq true) and (borrower [ord (bId)] eq mId) and
                   (nbLoans [ord (mId')] gt 0) and (nbLoans [ord (mId')] lt 2) and
                   (nil_reservation (reservation [ord (bId)]) eq true))
          end var
      end select
    [] Reserve (!mId, !bId, ?acquired, ?borrower, ?reservation, ?nbLoans)
          where ((acquired [ord (bId)] eq true) and
                 (borrower[ord (bId)] ne m⊥) and (borrower[ord (bId)] ne mId) and
                 (is_reserved (mId, reservation [ord (bId)]) eq false);
        select
            Take (!mId, !bId, ?acquired, ?borrower, ?reservation, ?nbloans)
              where ((acquired [ord (bId)] eq true) and
                     (borrower[ord (bId)] eq m⊥) and (nbLoans[ord (mId)] lt 2) and
                     (last_reservation (mId, reservation [ord (bId)]) eq true))
        [] Cancel (!mId, !bId)
        end select
    [] Return (!bId, ?acquired, ?borrower)
          where ((acquired [ord (bId)] eq true) and (borrower[ord (bId)] eq mId))
    [] var mId' : MID in
          mId' := any MID where ((mId' ne m⊥) and (mId' ne m⊥));
          Transfer (!mId', !bId, ?acquired, ?borrower, ?reservation, ?nbLoans)
            where ((acquired [ord (bId)] eq true) and (borrower [ord (bId)] eq mId) and
                   (nbLoans [ord (mId')] gt 0) and (nbLoans [ord (mId')] lt 2) and
                   (nil_reservation (reservation [ord (bId)]) eq true))
          end var
    end select
  end var
end process

process Main [Register, Lend, Take, Renew, Return, Reserve, Cancel, Unregister,
              Acquire, Discard : any] is
  par Register, Lend, Take, Renew, Return, Reserve, Cancel, Unregister, Acquire, Discard in
    par
      par
        loop L1 in select break L1 []
          book [Acquire, Discard] (b1)
        end select end loop
```

```
        ||
          loop L1 in select break L1 []
            book [Acquire, Discard] (b2)
          end select end loop
        end par
      ||
        par
          loop L1 in select break L1 []
            Register(m1);
            par
              loop L2 in select break L2 []
                member [Lend, Take, Renew, Return, Reserve, Cancel, Transfer] (m1, b1)
              end select end loop
            ||
              loop L2 in select break L2 []
                member [Lend, Take, Renew, Return, Reserve, Cancel, Transfer] (m1, b2)
              end select end loop;
            end par;
            var nbLoans : NB, bId : BID, reservation : RESERV in
              Unregister (m1, ?nbLoans, ?reservation)
                where ((nbLoans [ord (m1)] eq 1) and
                       (is_reserved (m1, reservation [ord (b1)]) eq false) and
                       (is_reserved (m1, reservation [ord (b2)]) eq false))
            end var
          end select end loop
        ||
          loop L1 in select break L1 []
            Register(m2);
            par
              loop L2 in select break L2 []
                member [Lend, Take, Renew, Return, Reserve, Cancel, Transfer] (m2, b1)
              end select end loop
            ||
              loop L2 in select break L2 []
                member [Lend, Take, Renew, Return, Reserve, Cancel, Transfer] (m2, b2)
              end select end loop
            end par;
            var nbLoans : NB, bId : BID, reservation : RESERV in
              Unregister (m2, ?nbLoans, ?reservation)
                where ((nbLoans [ord (m2)] eq 1) and
                       (is_reserved (m2, reservation [ord (b1)]) eq false) and
                       (is_reserved (m2, reservation [ord (b2)]) eq false))
            end var
          end select end loop;
        end par
    ||
      M [Acquire, Discard, Register, Lend, Take, Renew, Return, Reserve, Cancel, Unregister,
         Transfer]
    end par
  end process

  end module
```

C. MCL Formulas for Requirements R1 to R15

Requirement R1 "*A book can always be acquired by the library when it is not currently acquired*"

macro $R_1\,(B) =$
  (
    $[\,(\textbf{not}\ \{\texttt{ACQUIRE !}B\})^*\,]\ \langle\,\{\texttt{ACQUIRE !}B\}\,\rangle$ **true**
    and
    $[\,\textbf{true}^*\,.\,\{\texttt{DISCARD !}B\}\,.\,(\textbf{not}\ \{\texttt{ACQUIRE !}B\})^*\,]\ \langle\,\{\texttt{ACQUIRE !}B\}\,\rangle$ **true**
)
end_macro
$R_1\,(\texttt{"B1"})$ **and** $R_1\,(\texttt{"B2"})$ **and** $R_1\,(\texttt{"B3"})$

This is a classical liveness property. The second conjunct of "$R_1\,(B)$" expresses the eventuality that a book be withdrawn from the library before it is reacquired.

Requirement R3 "*An acquired book can be discarded only if it is neither borrowed nor reserved*"

$[\,\textbf{true}^*\,.\,\{?G : \textbf{string}\ \texttt{?any} : \textbf{string}\ ?B : \textbf{string}\ \textbf{where}\ (G = \texttt{"LEND"})\ \textbf{or}\ (G = \texttt{"TAKE"})\}\,.$
  $(\textbf{not}\ \{\texttt{RETURN !}B\})^*\,.\,\{\texttt{DISCARD !}B\}\,]$ **false**
and
$[\,\textbf{true}^*.\{\texttt{RESERVE ?any} : \textbf{string}\ ?B : \textbf{string}\}\,.$
  $(\textbf{not}\ (\{\texttt{CANCEL ?any} : \textbf{string}\ \texttt{!}B\}\ \textbf{or}\ \{\texttt{RETURN !}B\}))^*\,.\,\{\texttt{DISCARD !}B\}\,]$ **false**

Requirement R4 "*A person must be a member of the library in order to borrow a book*"

macro $R_4\,(M) =$
  (
    $[\,(\textbf{not}\ \{\texttt{JOIN !}M\})^*\,.\,(\{\texttt{LEND !}M\ \texttt{?any} : \textbf{string}\}\ \textbf{or}\ \{\texttt{TAKE !}M\ \texttt{?any} : \textbf{string}\})\,]$ **false**
    and
    $[\,\textbf{true}^*.\{\texttt{LEAVE !}M\}\,.\,(\textbf{not}\ \{\texttt{JOIN !}M\})^*\,.$
      $(\{\texttt{LEND !}M\ \texttt{?any} : \textbf{string}\}\ \textbf{or}\ \{\texttt{TAKE !}M\ \texttt{?any} : \textbf{string}\})\,]$ **false**
)
end_macro
$R_4\,(\texttt{"M1"})$ **and** $R_4\,(\texttt{"M2"})$ **and** $R_4\,(\texttt{"M3"})$

The first conjunct or "$R_4\,(M)$" expresses the fact that a member cannot borrow a book if (s)he has not registered to the library. The second conjunct expresses that if a member relinquishes his/her membership, (s)he may not lend a book neither via the regular loan action `LEND` nor the reservation action `RESERVE`.

Requirement R5 "*A book can be reserved only if it has been borrowed or already reserved by some member*"

macro $R_5\,(B) =$
  (
    [(**not** ({LEND ?**any** : **string** !B} **or** {TAKE ?**any** : **string** !B}))* .
      {RESERVE ?**any** : **string** !B}] **false**
    and
    [**true*** . {RETURN !B} .
      (**not** ({LEND ?**any** : **string** !B} **or** {TAKE ?**any** : **string** !B}))* . {RESERVE !B}] **false**
    and
    [(**not** ({LEND ?**any** : **string** !B} **or** {TAKE ?**any** : **string** !B} **or**
      {TRANSFER ?**any** : **string** !B} **or** {RESERVE ?**any** : **string** !B}))* .
        {RESERVE ?**any** : **string** !B}] **false**
)
end_macro
$R_5\,$("B1") **and** $R_5\,$("B2") **and** $R_5\,$("B3")

The first conjunct expresses the obligation for a book not to be lent in order to be added to the reservation list. The second conjunct complements the first in the sense that at least one loan cycle is completed in the beginning of the transition sequence via "{ RETURN !B }" thus making the book available for loan again. The third conjunct denies any reservation history for the book in question. All possible loan operations should be excluded as well.

Requirement R6 "*A book cannot be reserved by the member who is borrowing it*"

[**true***.{LEND ?M : **string** ?B : **string**} .
  (**not** ({RETURN !B} **or** {TRANSFER ?$M2$: **string** !B}))* . {RESERVE !M !B}] **false**

The difficulty here lies in the fact that the borrower may transfer the book to another member. For this reason, the following formula is false.

[**true*** . {LEND ?M : **string** ?B : **string**} . (**not** ({RETURN !B}))* . {RESERVE !M !B}] **false**

Requirement R8 "*A book cannot be lent to a member if it is reserved*"

macro $R_8\,(B, M_1, M_2) =$
  (
    [**true*** . {RESERVE !M_1 !B} . (**not** ({TAKE !M_1 !B} **or** {CANCEL !M_1 !B}))* .
      {LEND !M_2 !B}] **false**
)
end_macro
$R_8\,$("B1", "M1", "M2")

In this case (as well as for the subsequent requirements R9, R11, R13, R14, and R15), we only check the property for a specific book (B1) and members (M1, M2). So doing, we exploit the symmetry of the specification (all books and members have similar behaviour), which is crucial to avoid the exponential state space explosion.

Requirement R9 "*A member cannot renew a loan or give the book to another member if the book is reserved*"

macro $R_9\,(B, M) =$
  (
    $[\,\mathbf{true}^*\,.\,\{\texttt{RESERVE !}M\ \texttt{!}B\}\,.\,(\mathbf{not}\ (\{\texttt{TAKE !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{CANCEL !}M\ \texttt{!}B\}))^*\,.\,\{\texttt{RENEW !}B\}\,]\ \mathbf{false}$
)
end_macro
$R_9\,(\texttt{"B1"}, \texttt{"M1"})$

Requirement R10 "*A member is allowed to take a reserved book only if he owns the oldest reservation*"

$[$
  $\mathbf{true}^*\,.$
  $\{\texttt{RESERVE ?}M_1 : \mathbf{string}\ \texttt{?}B : \mathbf{string}\}\,.$
  $(\mathbf{not}\ (\{\texttt{TAKE !}M_1\texttt{!}B\}\ \mathbf{or}\ \{\texttt{CANCEL !}M_1\texttt{!}B\}\ \mathbf{or}\ \{\texttt{TRANSFER !}M_1\texttt{!}B\}))^*\,.$
  $\{\texttt{RESERVE ?}M_2 : \mathbf{string}\ \texttt{!}B\ \mathbf{where}\ M_2 \neq M_1\}\,.$
  $(\mathbf{not}\ (\{\texttt{TAKE !}M_1\texttt{!}B\}\ \mathbf{or}\ \{\texttt{CANCEL !}M_1\texttt{!}B\}\ \mathbf{or}\ \{\texttt{TRANSFER !}M_1\texttt{!}B\}))^*\,.$
  $\{\texttt{TAKE !}M_2\texttt{!}B\}$
$]\ \mathbf{false}$

This property has been rephrased in the following way: If two members reserve a book, the first member to get it, is the first to have ordered it.

Requirement R11 "*A book can be taken only if it is not borrowed*"

macro $R_{11}\,(B, M) =$
  (
    $[\,\mathbf{true}^*\,.\,(\{\texttt{LEND !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{TAKE !}M\ \texttt{!}B\})\,.\,(\mathbf{not}\ (\{\texttt{RETURN !}B\}))^*\,.$
      $(\{\texttt{LEND !}M\ \texttt{!}B\}\ \mathbf{or}\ \{\texttt{TAKE !}M\ \texttt{!}B\})\,.\,(\mathbf{not}\ (\{\texttt{RETURN !}B\}))^*\,.\,\{\texttt{RETURN !}B\}\,]\ \mathbf{false}$
)
end_macro
$R_{11}\,(\texttt{"B1"}, \texttt{"M1"})$

This property corresponds to the pattern "α_1 does not occur between α_2 and α_3", which is expressed by the following scheme, easily recognizable in this formula:

$$[\,\mathbf{true}^*\,.\,\alpha_2\,.\,(\mathbf{not}\ \alpha_3)^*\,.\,\alpha_1\,.\,(\mathbf{not}\ \alpha_3)^*\,.\,\alpha_3\,]\ \mathbf{false}$$

Requirement R13 "*A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or cancelled*"

macro $R_{13}\,(B, M) =$
  (
    [**true*** .
      ({LEND !M !B} **or** {TAKE !M !B}) .
      (**not** ({RETURN !B} **or** {TRANSFER !"M2" !B} **or** {TRANSFER !"M3" !B}))* .
      {LEAVE !M} . (**not** ({RETURN !B} **or** {TRANSFER !"M2" !B} **or** {TRANSFER !"M3" !B}))* .
      ({RETURN !B} **or** {TRANSFER !"M2" !B} **or** {TRANSFER !"M3" !B})] **false**
    and
    [**true*** . {RESERVE !M !B} . (**not** ({TAKE !M !B} **or** {CANCEL !M !B}))* .
      {LEAVE !M} . (**not** ({TAKE !M !B} **or** {CANCEL !M !B}))* .
      ({TAKE !M !B} **or** {CANCEL !M !B})] **false**
)
end_macro
R_{13} ("B1", "M1")

Requirement R14 "*Ultimately, there is always a procedure that enables a member to leave the library*"

macro $R_{14}\,(M) =$
  (
    [**true*** . {JOIN !M} . (**not** {LEAVE !M})*] ⟨ (**not** {LEAVE !M})* . {LEAVE !M} ⟩ **true**
)
end_macro
R_{14} ("M1")

Requirement R15 "*A member cannot borrow more than the loan limit defined at the system level for all users*"

macro $R_{15}\,(M) =$
  (
    [**true*** . **let** B_1 : **string** := "B1", B_2 : **string** := "B2" **in**
      ({LEND !M !B_1} **or** {TAKE !M !B_1}) .
      (**not** ({TRANSFER ?M_2 : **string** !B_1} **or** {RETURN !B_1}))* .
      ({LEND !M !B_2} **or** {TAKE !M !B_2}) **end let**] **false**
)
end_macro
R_{15} ("M1")

This property is dependent on the maximum number *NbLoans* of books a member can have at any time in his/her possession. In the above, *NbLoans* was set to two.