Translating EB3 to LNT for verification with CADP

Dimitris Vekris, Frederic Lang, Catalin Dima, Radu Mateescu

Translating EB^3 to LOTOS NT for verification with CADP

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

[1] LACL, Université Paris-Est
61 av. du Général de Gaulle, 94400 Créteil, France
{dimitrios.vekris,dima}@u-pec.fr
[2] CONVECS team, INRIA Grenoble Rhône-Alpes and LIG (Laboratoire d'Informatique de Grenoble)
655, av. de l'Europe, Montbonnot, F-38334 Saint Ismier, France
{Frederic.Lang,Radu.Mateescu}@inria.fr

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 entity types in a system, and attribute function definitions describing the entity attribute types. 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 of 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 EB^3 and the LNT specification. We automated this translation with the $EB^3$2LNT tool, which makes it possible to verify EB^3 specifications with the use of the CADP toolbox.

1 Introduction

The EB^3 [15] method is an event-based paradigm tailored for information systems (ISs). A typical EB^3 specification defines entities, associations, and their respective attributes. The process algebraic nature of EB^3 permits the explicit definition of intra-entity constraints. Yet its specificity against common state-space specifications, such as the B method [1] and Z, lies in the use of attribute functions, a special kind of recursive functions on the system trace, which combined with guards, facilitate the definition of complex inter-entity constraints involving the history of events. The use of attribute functions is claimed to simplify system understanding, enhance code modularity and streamline maintenance.

In this paper, we present part of our work regarding the verification of EB^3, i.e., the detection of errors inherent in EB^3 specifications. Specification errors in EB^3 can be detected with the aid of invariants also known as static properties or temporal properties known as dynamic properties. From a state-based point of view, an invariant describes a property on state variables that must be preserved by each transition or event. A dynamic property relates several events. Tools such as Atelier B [10] provide methodologies on how to define and prove invariants. In

[18], an automatic translation of EB^3's attribute functions with B is attempted. Although the B Method [1] is suitable for specifying static properties, dynamic properties are very difficult to express and verify in B. Hence, in our attempt to verify dynamic properties of EB^3 specifications we move our attention to model-checking techniques.

The verification of EB^3 specifications against temporal properties with the use of model checking has been the subject of some work in the recent years. [14] compares six model checkers for the verification of IS case studies. The specifications used in [14] derive from industrial case studies, but the prospect of a uniform translation from EB^3 program specifications is not studied. [7] summarizes the fundamental difficulties for designing a compiler that would translate a given EB^3 specification to process algebra LOTOS NT [11], an approach that would make use of the verification suite CADP (*Construction and Analysis of Distributed Processes*) [16] to verify properties against the source system. In short, the majority of these works treat specific case studies drawn from the information systems domain leading to ad-hoc verification translations, but nonetheless lacking in generalization capability.

The absence of model-checking tools for the verification of EB^3 specifications led us to consider the direct casting of EB^3 to model checking specifications. This approach turned out to be rather involved. We used the Nu-SMV [8] model checker as our target language. Nu-SMV verifies models against branching time properties in CTL [9]. The need to verify EB^3, that is event-based by nature, against action-based properties, entailed some "syntactic acrobatics" from our side, as a means to transform them to their state-based equivalent properties. The issue is addressed in [12]. Furthermore, Nu-SMV's close resemblance to low-level programming languages made the automatic translation a strenuous and time-consuming task. The treatment of inherent features of EB^3, such as the non-deterministic closure of EB^3 expressions, i.e., E^* and the guarded expressions containing attribute function calls, necessitated the use of intermediate structures such as Petri Nets [27]. The lack of recursive type definitions within the Nu-SMV specification language forced us to consider binary encodings for complex data structures in EB^3 such as lists and sets. Finally, the translation algorithm gave us complex Nu-SMV specifications for simple EB^3 specifications and poor verification performance.

We then studied the option of using existing model checking tools for process algebras. One such tool is FDR [5] that verifies CSP [19] specifications with the method of refinement checking between the system and the property in question expressed as a CSP model. The design of EB^3 was partly inspired by CSP, which made FDR a promising candidate for this task. However, the refinement-based approach to verification does not seem very well suited for verifying certain temporal properties as explained in [21]. Though the majority of the properties we are interested in could well be verified in FDR, we were soon discouraged by our experimental results, which proved that it was unnatural to translate EB^3 specifications to CSP.

Finally, we turned our attention to LOTOS NT (LNT for short), a process algebra specification that derived from LOTOS [4]. It is one of the input languages of CADP, a toolbox with state-of-the-art verification features. CADP permits the verification of system specifications against action-based temporal properties. Interesting projects in the area of protocol verification with CADP can be found in [6] and [31], among many others[3].

Translating process algebra specifications to LNT is not a new idea. In [24], an automatic translation of π-calculus [26] to LNT was undertaken. In [30], FSP [23] specifications were translated to LOTOS. The aim for both projects was to make the verification features of CADP available to π-calculus and FSP users. Another interesting project is found in [17], in which CHP (Communicating Hardware Processes), a process algebra devoted to the description of asynchronous hardware processes, is translated to LOTOS. To this end, we propose an automatic translation from EB^3 specifications to LNT. We identify the difficulties mentioned in [7] and solve them. In Section 2 and 3, we introduce EB^3 and LNT. Section 4 describes the $EB^3$2LNT translator. Section 5 illustrates an industrial case study that provides evidence to the usefulness of our translation and Section 6 draws up some conclusions and lines for future work.

2 EB^3

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

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

We limit the presentation to the process algebra (see Figure 1) and the set of attribute functions used in the IS. An EB^3 Specification is a set of attribute function definitions, *AttrF* and a set of process definitions, *ListPE*. The trace semantics of EB^3, Sem_{T} [15], is given in Fig. 2 as a set of rules named $R_{\mathsf{T}}-1$ to $R_{\mathsf{T}}-11$.

Let $\rho \in Act$ stand for an action of either form $\alpha\,(p_1 : T_1, \ldots, p_n : T_n)$, where $\alpha \in lab$ is the *label* of the action and p_i $(i \in 1..n)$ are elements of type T_i, or λ, which stands for the internal action. An action ρ is the simplest EB^3 process, whose semantics are given by rule $R_{\mathsf{T}}-1$. Note that λ is not visible in the EB^3 execution trace, i.e., it does not impact the definition of attribute functions. The

[3] see http://cadp.inria.fr/case-studies for a list of 152 published case-studies using CADP

$$
\begin{array}{rl}
EB^3 ::= & AttrF;\ ListPE \\
AttrF ::= & \ldots /^* \text{ attribute functions } ^*/ \\
ListPE ::= & P(\overline{x}) = E \mid P(\overline{x}) = E\ ;\ ListPE \\
E ::= & \surd \mid \lambda \mid \alpha(\overline{v}) \mid E.E \mid E|E \mid E^* \mid E|[\Delta]|E \mid |x{:}V{:}E \mid \\
& |[\Delta]|x{:}V{:}E \mid GE \Rightarrow E \mid P(\overline{t})
\end{array}
$$

Fig. 1. EB^3 syntax

$$
R_{\mathsf{T}}-1 : \frac{}{\rho \xrightarrow{\rho} \surd} \qquad
R_{\mathsf{T}}-2 : \frac{E_1 \xrightarrow{\rho} E_1'}{E_1.E_2 \xrightarrow{\rho} E_1'.E_2}
$$

$$
R_{\mathsf{T}}-3 : \frac{E \xrightarrow{\rho} E'}{\surd.E \xrightarrow{\rho} E'} \qquad
R_{\mathsf{T}}-4 : \frac{E_1 \xrightarrow{\rho} E_1'}{E_1|E_2 \xrightarrow{\rho} E_1'}
$$

$$
R_{\mathsf{T}}-5 : \frac{}{E^* \xrightarrow{\lambda} \surd} \qquad
R_{\mathsf{T}}-6 : \frac{E \xrightarrow{\rho} E'}{E^* \xrightarrow{\rho} E' \cdot E^*}
$$

$$
R_{\mathsf{T}}-7 : \frac{}{\surd|[\Delta]|\surd \xrightarrow{\lambda} \surd} \qquad
R_{\mathsf{T}}-8 : \frac{E_1 \xrightarrow{\rho} E_1' \qquad E_2 \xrightarrow{\rho} E_2'}{E_1|[\Delta]|E_2 \xrightarrow{\rho} E_1'|[\Delta]|E_2'}\, in(\rho, \Delta)
$$

$$
R_{\mathsf{T}}-9 : \frac{E \xrightarrow{\rho} E'}{GE \Rightarrow E \xrightarrow{\rho} E'}\, \|GE\| \qquad
R_{\mathsf{T}}-10 : \frac{E_1 \xrightarrow{\rho} E_1'}{E_1|[\Delta]|E_2 \xrightarrow{\rho} E_1'|[\Delta]|E_2}\, \neg in(\rho, \Delta)
$$

$$
R_{\mathsf{T}}-11 : \frac{E[\overline{x} := \overline{t}] \xrightarrow{\rho} E'}{P(\overline{t}) \xrightarrow{\rho} E'} P(\overline{x}) = E
$$

Fig. 2. EB^3 trace semantics Sem_{T}

symbol $\surd$ denotes successful execution. EB^3 processes can be combined with classical process algebra operators such as the *sequence* ($R_{\mathsf{T}}-2, 3$), the *choice* ($R_{\mathsf{T}}-4$) and the *Kleene* Closure ($R_{\mathsf{T}}-5, 6$) operators. Rules ($R_{\mathsf{T}}-7, 8, 10$) refer to the *parallel* composition $E_1|[\Delta]|E_2$ of E_1, E_2 with synchronization on $\Delta \subseteq lab$. 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. Expression $E_1|||E_2$ is equivalent to $E_1|[\emptyset]|E_2$ and $E_1||E_2$ to $E_1|[lab]|E_2$.

In $R_{\mathsf{T}}-9$, the *guarded expression* process $GE \Rightarrow E$ can execute E if the predicate GE holds. GE contains calls to attribute functions, i.e., functions defined on the system trace. Concretely, the truth value of GE depends on the executed actions.

Quantification is permitted for *choice* and *parallel* composition. If V is a set of attributes $\{t_1, \ldots, t_n\}$, $|x{:}V{:}E$ and $|[\Delta]|x{:}V{:}E$ stand respectively for $E[x := t_1]| \ldots |E[x := t_n]$ and $E[x := t_1]|[\Delta]| \ldots |[\Delta]|E[x := t_n]$, where $E[x := t]$ denotes

1. A book can be acquired by the library. It can be discarded, but only if it has not been lent.
2. An individual must join the library in order to borrow a book.
3. A member can relinquish library membership only when all his loans have been returned.
4. A member cannot borrow more than the loan limit defined at the system level for all users.

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

$nbLoans\,(\mathsf{T} : tr, mId : MID) : Nat_\bot\ =$ **match** T **with** $[\,] \rightarrow \bot$ $\mid \mathsf{T}'.\ Lend\,(bId, mId) \rightarrow nbLoans\,(\mathsf{T}', mId) + 1$ $\mid \mathsf{T}'.\ Register\,(mId) \rightarrow 0$ $\mid \mathsf{T}'.\ Unregister\,(mId) \rightarrow \bot$ $\mid \mathsf{T}'.\ Return\,(bId) \wedge mId = borrower\,(\mathsf{T}, bId)$ $\rightarrow nbLoans\,(\mathsf{T}', mId) - 1$ $\mid \_ \rightarrow nbLoans\,(\mathsf{T}', mId)$	$borrower\,(\mathsf{T} : tr, bId : BID) : MID\ =$ **match** T **with** $[\,] \rightarrow \bot$ $\mid \mathsf{T}'.\ Lend\,(bId, mId) \rightarrow mId$ $\mid \mathsf{T}'.\ Return\,(bId) \rightarrow \bot$ $\mid \_ \rightarrow borrower\,(\mathsf{T}', bId)$

Fig. 3. EB^3 specification and attribute function definitions

the replacement of all occurrences of x by t. For instance, $||x : \{1,2,3\} : a\,(x)$ stands for $a\,(1)||a\,(2)||a\,(3)$. By convention,$|x{:}\emptyset{:}E = |[\Delta]|x{:}\emptyset{:}E = \surd$. Given an EB^3 process expression E, we write $vars\,(E)$ for the set of variables occurring free in E.

Example. In Fig. 3, we give the functional requirements of a library management system and the corresponding EB^3 Specification.

Function *main* is the parallel interleaving between m instances of process *book* and p instances of process *member*. Process *book* stands for a book acquisition followed by its eventual discard.

The attribute function $borrower\,(\mathsf{T},\ bId)$, where T is the current trace, returns the current borrower of book bId or $\bot$ if the book is not lent, by looking for events of the form $Lend\,(mId, bId)$ or $Return\,(bId)$ in the trace. In process book, action $Discard\,(bId)$ is thus guarded to guarantee that book bId cannot be discarded if it is currently lent.

Execution. We show how the EB^3 Specification describing the library management system is evaluated w.r.t. the standard semantics, Sem_{T} and alternative memory-based semantics, Sem_{M}[4], wherein attribute functions are computed during program evolution and stored into program memory M. Sem_{M} and Sem_{T} can be proven equivalent. The proof is beyond the scope of this paper.

The idea behind Sem_{M} lies in the simple observation that the attribute functions can be turned into state variables carrying the effect of the system trace on their corresponding values. For instance, provided that $BID = \{b_1, b_2\}$, $MID = \{m_1, m_2\}$, we take state variables $M = (bor[bId1], bor[bId2], nbL[mId1], nbL[mId2])$,

[4] Sem_{M} can be found in the appendix for referee's eyes only

where *bor* stands for *borrower* and *nbL* for *nbLoans*, respectively. More formally, if $f\,(\mathsf{T}, x_1{:}T_1, \ldots, x_r{:}T_r)$ is an attribute function, we construct $|T_1| \times \ldots \times |T_r|$ state variables, where $|T_i|$ $(i \in 1..r)$ stands for T_i's cardinality. Intuitively, coding attribute functions as part of the system state is beneficial from a model-checking point of view as as it avoids keeping (potentially huge) trace in memory.

We set $NbLoans = 2$. Fig. 4 shows how *main* is modified for the valid trace: $\mathsf{T}_D = Lend\,(bId1, mId1).Reg\,(mId2).Reg\,(mId1).Acq\,(bId2).Acq\,(bId1)$[5].

$main$ (A)

$\xrightarrow{Acq\,(bId2).Acq\,(bId1)}$

$borrower\,(\mathsf{T},\ bId1) = \bot \rightarrow Discard\,(bId1)\ |||$
$borrower\,(\mathsf{T},\ bId2) = \bot \rightarrow Discard\,(bId2)\ |||$
$(\ ||| \ mId : MID : member\,(mId)*\ )$ (B)

$\xrightarrow{Reg\,(mId2).Reg\,(mId1)}$

$borrower\,(\mathsf{T},\ bId1) = \bot \rightarrow Discard\,(bId1)\ |||$
$borrower\,(\mathsf{T},\ bId2) = \bot \rightarrow Discard\,(bId2)\ |||$
$(\ ||| \ bId : BID : loan\,(mId1, bId)*\ ).\ Unregister\,(mId1).\ member\,(mId1)*\ |||$
$(\ ||| \ bId : BID : loan\,(mId2, bId)*\ ).\ Unregister\,(mId1).\ member\,(mId2)*$ (C)

$\xrightarrow{Lend\,(bId1,\ mId1)}$

$borrower\,(\mathsf{T},\ bId1) = \bot \rightarrow Discard\,(bId1)\ |||$
$borrower\,(\mathsf{T},\ bId2) = \bot \rightarrow Discard\,(bId2)\ |||$
$(Return\,(bId1).\ loan\,(mId1, bId1)*\ ||| \ loan\,(mId1, bId2)*).Unregister\,(mId1).\ member\,(mId1)*\ |||$
$(\ ||| \ bId : BID : loan\,(mId2, bId)*\ ).\ Unregister\,(mId1).\ member\,(mId2)*$ (D)

Fig. 4. Execution

	T	$M = (bor[bId1], bor[bId2], nbL[mId1], nbL[mId2])$
A	$[\,]$	$(\bot, \bot, \bot, \bot)$
B	$Acq\,(bId2).Acq\,(bId1)$	$(\bot, \bot, \bot, \bot)$
C	$\mathsf{T}_B.Reg\,(mId2).Reg\,(mId1)$	$(\bot, \bot, 0, 0)$
D	$\mathsf{T}_C.Lend\,(bId1,\ mId1)$	$(mId1, \bot, 1, 0)$

Fig. 5. States

The intermediate states A, B, C and D for Sem_{T} and Sem_{M} are given in Fig. 5. The first column corresponding to Sem_{T} keeps track of the system trace, whereas Sem_{M} gives a finite state system since the domains of its attribute functions are finite and bounded. All variables are equal to $\bot$[6] for $\mathsf{T} = [\,]$. Focussing on transition $C \rightarrow D$, in order to check $borrower\,(\mathsf{T}, bId1) = \bot \wedge nbLoans\,(\mathsf{T}, mId1) <$

[5] *Acq* stands for *Acquire* and *Reg* for *Register*, respectively
[6] see *borrower*'s and *nbLoans*'s script for $\mathsf{T} = [\,]$ in Fig. 3

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

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

2, Sem_{T} evaluates $borrower\,(\mathsf{T}, bId1)$ and $nbLoans\,(\mathsf{T}, mId1)$ by traversing the trace and applying their corresponding attribute function formulas. On the contrary, Sem_{M} evaluates M based solely on the current memory and the event to occur. We have $bor_D[bId1] = next\,(bor_C[bId1]) = mId1$[7], $bor_D[bId2] = bor_C[bId2] = \perp$, $nbL_D[mId1] = nbL_C[mId1] + 1 = 1$ and also $nbL_D[mId2] = 0$, if the event $Lend\,(bId1, mId1)$ is to be executed.

3 LNT

LNT combines 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 an LNT specification.

We present the fragment of LNT that serves as the target of our translation. Its syntax is given in Fig. 6. LNT terms denoted by B 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 G with bidirectional transmission of multiple values. Synchronizations may also contain optional guards (**where**) expressing boolean conditions on received values. The special action δ is used for defining the semantics of sequential composition. The internal action is denoted by the special gate i, which cannot be used for synchronization. The parallel composition operator allows multiway rendezvous on the same gate. Expressions E are built from variables, type constructors, function applications and constants. Labels L identify loops, which can be stopped using "break L" from inside the loop body. Offer O can be either a send offer (!) or a receive offer (?). Processes are parameterized by gates and data variables. The semantics of LNT are formally defined in [11].

4 Translation from EB^3 to LNT

Attribute functions. Our translation of EB^3 relies on semantics Sem_{M}. Thus, we explicitly model in LNT a memory, which stores the state variables corre-

[7] see *borrower*'s script for $\mathsf{T} = \mathsf{T}'.Lend\,(bId, mId)$ in Fig. 3

sponding to attribute functions (we call these variables *attribute variables*) and is modified each time an action is executed.

A difficulty comes from the fact that LNT does not feature built-in global variables and that accesses to local variables is restricted in parallel processes, so that in "**par** $proc_1$ || $proc_2$ **end par**", every variable written in $proc_1$ cannot be accessed (i.e., read or written) in $proc_2$.

Therefore, assuming n attribute functions $f_1, \ldots, f_n$, we model the memory as a process M placed in parallel with the rest of the system (a common approach in process algebra), which manages for each attribute function f_i an attribute variable (also named f_i) that encodes the function. To read the values of these attribute variables (i.e., to evaluate the attribute functions), processes need to communicate with the memory M, and every action must have an immediate effect on the memory (so as to reflect the immediate effect on the execution trace). To achieve this, the memory process synchronizes with the rest of the system on every possible action of the system (including λ, to which we associate an LNT gate written *LAMBDA*), and updates its attribute variables accordingly. The list of attribute variables $\overline{f} = (f_1, \ldots, f_n)$ is added as a supplementary offer on each EB^3 action $\alpha\,(\overline{v})$, so that attribute variables can be directly accessed to evaluate the guard associated to the action, wherever needed. Therefore, every action $\alpha\,(\overline{v})$ will be encoded in LNT as $\alpha\,(!\overline{v}, ?\overline{f})$, and synchronised with an action of the form $\alpha\,(?\overline{x}, !\overline{f})$ in the memory process M, thus taking benefit of the bidirectional value exchange available in LNT, a feature inherited from LOTOS.

To formalize our translation, we first assume that $lab = \{\alpha_1, \ldots, \alpha_q\}$ (not including λ), $AtFct = \{f_1, \ldots, f_n\}$, each action α_j has formal parameters $\overline{x_j}$, and each attribute functions f_i has formal parameters $\overline{y}_i$. It is also assumed that attribute functions are defined following the scheme of Fig. 7, where $exp_i^{j,k}$ are expressions (of the same type as f_i's return type), $cond_i^{j,k}$ are boolean expressions, $hd\,(\mathsf{T})$ denotes the last element of the trace, and $tl\,(\mathsf{T})$ denotes the trace without its last element. Expressions can be constructed from objects and operations of user-defined domains, such as integers, booleans and more complex domains that we do not give formally, for lack of space. These objects and operations can be easily defined in LNT.

We also assume that the attribute functions are ordered, so that for all $h \in 1..n, i \in 1..n, j \in 1..q, k \in 1..m_j$, every function call of of the form $f_h\,(\mathsf{T}, \ldots)$ occurring in $exp_i^{j,k}$ or $cond_i^{j,k}$ satisfies $h < i$ and every call of the form $f_h\,(tl(\mathsf{T}), \ldots)$ satisfies $h \geq i$. 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. Ordering attribute functions in this way allows the memory to be updated consistently, from f_1 to f_n in turn. At every instant, not-yet-updated values correspond to calls of the form $f_h\,(tl(\mathsf{T}), \ldots)$ (the value of f_h on the previous trace), whereas already-updated values correspond to calls of the form $f_h\,(\mathsf{T}, \ldots)$ (the value of f_h on the current trace). The condition thus enables the trace parameter to be discharged from function calls, ensuring that while updating f_i, calls to functions f_h with $h < i$

$$f_i\,(\mathsf{T}, \overline{y}_i) = \begin{cases} exp_i^{0,0} & \text{if } \mathsf{T} = [\,] \\ \bigvee_{j=1}^{q} \bigvee_{k=1}^{m_j} (hd\,(\mathsf{T}) = \alpha_j\,(\overline{x}_j) \wedge cond_i^{j,k}) \Rightarrow exp_i^{j,k} & \text{otherwise} \end{cases}$$

Fig. 7. Attribute function definitions

correspond (now implicitly) to calls with parameter T, and calls to functions f_h with $h \geq i$ correspond to calls with parameter $tl(\mathsf{T})$.

```
process M [α_1, ..., α_q, LAMBDA : any] is
  var f : type(f), y_1 : type(y_1), ..., y_n : type(y_n), x_1 : type(x_1), ..., x_q : type(x_q) in
    upd_1^0; ... ; upd_n^0;
    loop
      select
      α_1 (?x_1, !f); upd_1^1; ... ; upd_n^1
      [] ... []
      α_q (?x_q, !f); upd_1^q; ... ; upd_n^q
      [] LAMBDA (!f)
    end select
    end loop
  end var
end process
upd_i^j ≐ enum (y_i, upd_i^{j,1}; ...; upd_i^{j,k})
upd_i^{j,k} ≐ if mod (cond_i^{j,k}) then f_i[ord (x_i)] := mod (exp_i^{j,k}) end if
enum ([], B) ≐ B
enum (x :: y, B) ≐ x := first_T;
                   loop L_x in
                     enum (y, B)
                     if x ≠ last_T then x := next_T (x) else break L_x end if
                   end loop where T = type (x)
[ord (x)] ≐ [ord (x_1)] ... [ord (x_l)], ?x = (?x_1, ..., ?x_l), where x = (x_1, ..., x_l)
mod (E) ≐ E [ f_i (tr, v_i) ← f_i[ord (v_i)] | i ∈ 1..n, tr ∈ {T, tl (T)} ]
```

Fig. 8. LNT code for the memory process implementing attribute functions

Process M is defined in Fig. 8. It runs an infinite loop, which "listens" to all possible actions α_j of the system. Each attribute variable f_i is an array with n_i dimensions, where n_i is the arity of the attribute function f_i minus 1 (because the trace parameter has been discharged). Dimension j thus corresponds to parameter y_i^j, so that $f_i[\mathbf{ord}\,(v_1)] \ldots [\mathbf{ord}\,(v_{n_i})]$ encodes the current value of $f_i\,(\mathsf{T}, v_1, \ldots, v_{n_i})$, where $\mathbf{ord}\,(v)$ is a predefined LNT function that denotes the *ordinate* of value v, i.e., a unique number between 0 and the cardinal of v's type minus 1. For each type T we assume that we have functions $first_T$ that

returns the first element of type T, $last_T$ that returns the last element of type T, and $next_T(x)$ that return the successor of x in type T. Such functions are available in LNT for all finite types. Function *mod* transforms an expression E by syntactically replacing function calls by array accesses, while discharging the trace parameter as explained above.

Upon synchronisation on action $\alpha_j\,(?\overline{x}_j, !\overline{f})$ with the LNT process corresponding to EB^3's *main* process (see Subsection 4.2), the values of all attribute variables f_i ($i \in 1..n$) are updated. Function upd_i^j casts the attribute function definition f_i to LNT. For $j = 0$, we get the initialisation function for f_i, i.e., the value of f_i when $\mathsf{T} = [\,]$. By convention, $cond_i^{0,0}$ is trivially true.

Process expressions. 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 correspondance in LNT. The main difficulty arises in the translation of guarded process expressions of the form $GE \Rightarrow E$, since rule $R_{\mathsf{T}}-9$ (Fig. 2, page 4) stipulates that GE is evaluated only on the first action ρ occurring in E. This led us to consider a second parameter for the translation function t, namely the guard GE to be evaluated on the first action occurring in the process expression E. The definition of $t\,(E, GE)$ is given in Figure 9. An EB^3 specification E_0 will then be translated into the LNT parallel composition "**par** $\alpha_1, \ldots, \alpha_q, LAMBDA$ **in** $t\,(E_0, \text{true})$ $||$ $M[\alpha_1, \ldots, \alpha_q]$ **end par**" and every process definition of the form "$P\,(\overline{x}) = E$" will be translated into the following LNT process:

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

The rules of Fig. 9 can be commented as follows:

- Rule (lam) 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 GE, 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 GE 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 Fig. 8. Rule (act) is similar.
- Rule (seq) translates EB^3 sequential composition into LNT sequential composition. By doing so, the evaluation of GE is passed to the first process expression.
- Rule (guard) makes a conjunction between the guard of the current process expression with the guard already accumulated from the context.
- Rules (sel) and (xsel) translate the choice and quantified choice operators of EB^3 into their direct LNT equivalent.
- Rule (loop) translates the Kleene closure into a combination of LNT loop and select, following the identity $E^* = E.E^*|\lambda$. Note however that this rule only applies to the case of a guard that is not trivially true. In other cases, Rule (other) applies, as will be commented below.

$$
\begin{array}{rll}
t\,(\lambda, GE) = & LAMBDA\,(?\overline{f})\ \textbf{where}\ mod\,(GE) & \text{(lam)} \\
t\,(\alpha\,(\overline{v}), GE) = & \alpha\,(\overline{v}, ?\overline{f})\ \textbf{where}\ mod\,(GE) & \text{(act)} \\
t\,(E_1.E_2, GE) = & t\,(E_1, GE);\ t\,(E_2, \text{true}) & \text{(seq)} \\
t\,(GE' \Rightarrow E, GE) = & t\,(E, GE\ \textbf{and}\ GE') & \text{(guard)} \\
t\,(E_1|E_2, GE) = & \textbf{select}\ t\,(E_1, GE)\ \texttt{[]}\ t\,(E_2, GE)\ \textbf{end select} & \text{(sel)} \\
t\,(|x\!:V\!:E, GE) = & \textbf{var}\ x := \textbf{any}\ V;\ t\,(E, GE)\ \textbf{end var} & \text{(xsel)} \\
t\,(E^*, \text{true}) = & \textbf{loop}\ L\ \textbf{in} & \\
& \quad \textbf{select} & \\
& \quad\quad t\,(E, \text{true})\ \texttt{[]}\ LAMBDA\,(?\overline{f});\ \textbf{break}\ L & \\
& \quad \textbf{end select} & \\
& \textbf{end loop} & \text{(loop)} \\
t\,(E_1|[\Delta]|E_2, \text{true}) = & \textbf{par}\ \Delta\ \textbf{in}\ t\,(E_1, \text{true})\ ||\ t\,(E_2, \text{true})\ \textbf{end par} & \text{(par)} \\
t\,(|[\Delta]|x\!:V\!:E, \text{true}) = & \textbf{par}\ \Delta\ \textbf{in}\ E[x \leftarrow v_1]\ ||\ \ldots\ ||\ E[x \leftarrow v_n]\ \textbf{end par} & \\
& \quad \text{where}\ V = \{v_1, \ldots, v_n\} & \text{(xpar)} \\
t\,(P\,(\overline{v}), \text{true}) = & P\,[\alpha_1, \ldots, \alpha_q, LAMBDA]\,(\overline{v}) & \text{(proc)}
\end{array}
$$

In all other cases:

$$
t\,(E, GE) = \begin{cases}
\textbf{if}\ mod\,(GE)\ \textbf{then}\ t\,(E, \text{true})\ \textbf{else stop end if} & \\
\quad \text{if } GE \text{ does not call attribute functions} & \\
\textbf{par}\ \alpha_1, \ldots, \alpha_q, LAMBDA\ \textbf{in} & \text{(other)} \\
\quad t\,(E, \text{true})\ ||\ pr_{GE}\,[\alpha_1, \ldots, \alpha_q]\,(vars\,(GE)) & \\
\textbf{end par} \qquad \text{otherwise} &
\end{cases}
$$

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

– Rule (par) translates EB^3 parallel composition into LNT parallel composition. Again, this rule only applies to the case of a guard that is not trivially true.
– Rule (xpar) 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. Again, this rule only applies to the case of a guard that is not trivially true.
– Rule (other) applies when the expression E is a process call or a Kleene closure or a parallel composition and the expression E is not trivially true.
If the guard GE does not use attribute functions, then its value does not depend on the trace, and it can be evaluated immediately without any communication with the memory process (first case).
Otherwise, the guard must apply only to the first action reached during execution of E. A way to implement this would consist in expanding the process expression E into a choice in which every branch has a fixed initial action, which the guard could apply to (a form commonly called *head normal form* [2] and used principally in the context of the process algebra ACP [3] as a means to analyse the behaviour of recursive process algebra definitions). We preferred an alternative solution that avoids the potential combinatorial explosion of code due to static expansion. A process pr_{GE} (defined in Fig. 10) is placed in parallel to $t\,(E, \mathrm{true})$ and both processes synchronise on all actions. Process pr_{GE} thus imposes on $t\,(E, \mathrm{true})$ the constraint that the first executed action must satisfy the condition GE that pr_{GE} encodes (**then** branch). For subsequent actions, the condition is relaxed (**else** branch).

Correctness of the Transformation.

Proposition 1. *Consider an IS, its EB^3 Specification and a set of actions Act. Then, for action $\rho \in Act$ and E,E' valid EB^3 processes in the IS:*

$$E \xrightarrow{\rho\,(\overline{x})} E' \text{ iff } \exists f \text{ such that } t\,(E, true) \xrightarrow{\rho\,(\overline{x}, f)} t\,(E', true)$$

The proof of this proposition is tedious and can be done by induction on the EB^3 processes.

Tool and Benchmark. 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 examined in its simplest version in Section 2 and a bank account management system.

Complexity of the Transformation. We noticed that for each EB^3 specification the equivalent LNT specification is twice as big. This expansion is caused firstly by the fact that all parallel compositions in EB^3 should be expanded in LNT and secondly by the fact that LNT is slightly more verbose and structured than EB^3. LNT requires more keywords, gates have to be declared explicitly and passed as parameters to each process call. In the worst case, the size of a LNT term is proportional to $O\,(\mathbf{max}\{|V_i|\})$, where $V_i, i \geq 0$ is the set of entity type

```
process pr_GE[α_1, ..., α_q : any] (z_1, ..., z_m) is
var b : bool, y_1 : type(y_1), ..., y_q : type(y_q) in
  b := true;
  loop L in select
    if b then
      b := false;
      select
        α_1 (?y_1, ?f) where mod(GE)
        [] ... []
        α_q (?y_q, ?f) where mod(GE)
        []
        LAMBDA (?f) where mod(GE)
      end select
    else
      select
        α_1 (?y_1, ?f)
        [] ... []
        α_q (?y_q, ?f)
        []
        LAMBDA (?f)
      end select
    end if
  [ ] break L end select end loop
end var
end process
```

Fig. 10. Process pr_{GE}

domains present in processes of the form $x : V_i : E$ and $|[\Delta]|x : V_i : E$. $|V_i|$ stands for the cardinality of V_i.

5 Case Study

The case study examined here is the library management system seen in Section 2 enhanced wih extra functionalities:

1. A book can always be acquired by the library when it is not currently acquired.
2. A book cannot be acquired by the library if it is already acquired.
3. An acquired book can be discarded only if it is neither borrowed nor reserved.
4. A person must be a member of the library in order to borrow a book.
5. A book can be reserved only if it has been borrowed or already reserved by some member.
6. A book cannot be reserved by the member who is borrowing it.
7. A book cannot be reserved by a member who is reserving it.
8. A book cannot be lent to a member if it is reserved.
9. A member cannot renew a loan or give the book to another member if the book is reserved.
10. A member is allowed to take a reserved book only if he owns the oldest reservation.
11. A book can be taken only if it is not borrowed.
12. A member who has reserved a book can cancel the reservation at anytime before he takes it.
13. A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or canceled.
14. Ultimately, there is always a procedure that enables a member to leave the library.

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

Verification with CADP. All 15 requirements (that we name $p1-15$) to verify were expressed in MCL [25]. MCL is an extension of the alternation-free μ-calculus [20] with ACTL-like [13] action formulas and PDL-like [22] regular expressions, allowing a concise and intuitive description of safety, liveness, and fairness properties without sacrificing the efficiency of verification. MCL combines data handling mechanisms (quantified variables and fixed point parameters), extended regular expressions, and constructs inspired from programming languages.

We verified the LNT specification corresponding to the library management system on an Intel(R) Core(TM) i7 CPU 880 @ 3.07GHz. All properties were proven true as expected and the results can be found in Fig. 11. In the first line, (b, m) corresponds to the number of books and members of the IS. The second line gives the time needed to generate the corresponding LTS to the LNT model and the rest lines give the verification time for each requirement. Property $p2$ is formalised by the following MCL formula:

```
[true*. { ACQ !"B1" }. (not { DIS !"B1" })*. { ACQ !"B1" }] false
```

This formula follows the standard *safety* pattern: "$[\alpha]$false". It evaluates to true if no execution path matches the regular expression written inside the box modality. The meaning of $p2$'s regular expression is that we cannot have a sequence of ACQUIRE operations for book B1, if there is no DISCARD operation for B1 in the meantime. Notation true inside [ ] refers to all possible actions in the IS. Property $p12$ is formalised in the following manner:

```
[ true*. { RES !"M1" !"B1" }.
    ( not ({ TAKE !"M1" !"B1" } or { TRANSFER !"M1" !"B1" } ))* ]
⟨( not ( { TAKE !"M1" !"B1" } or { TRANSFER !"M1" !"B1" } ))*.
    { CANCEL !"M1" !"B1" }⟩ true
```

This formula is a *liveness* property. Liveness properties of the form "$[\alpha]\langle\beta\rangle$true" state that every execution path matching the regular expression α (in this case, book B1 has been reserved by member M1 and subsequently neither taken not transfered) ends in a state from which there exists an execution path matching the regular expression β (in this case, the reservation can be cancelled before being taken or transfered).

6 Conclusion

We have presented a translation from EB^3 specifications to LNT. This translation makes it possible to use all the state-of-the-art verification features of CADP to analyze EB^3 specifications. The translation is automated by $EB^3$2LNT and validated on various examples.

We plan to present a formal proof of correctness for the transformation. We will also study abstraction techniques for the verification of properties regardless of the number of components e.g. members, books that participate in the IS (Parameterized Model Checking). We will observe how the insertion of new functionalities to the ISs affects this issue. Finally, we will formalize this in the context of EB^3 specifications.

(b,m)	(3,2)	(3,3)	(3,4)	(4,3)
time	1.892s	14.421s	31m39.743s	140m22.734s
p1	0.328s	1.800s	5m19.108s	20m13.876s
p2	0.208s	2.960s	9m26.623s	36m7.443s
p3	0.236s	4.960s	12m52.984s	55m33.744s
p4	0.240s	1.716s	5m15.644s	18m40.526s
p5	0.268s	2.188s	6m46.537s	21m52.770s
p6	0.216s	1.900s	5m53.798s	19m40.458s
p7	0.232s	2.224s	6m45.353s	22m39.589s
p8	0.224s	2.240s	6m52.046s	22m27.872s
p9	0.244s	2.288s	6m38.593s	22m29.164s
p10	0.280s	13.345s	43m59.497s	62m7.837s
p11	0.260s	2.544s	6m36.161s	22m14.027s
p12	0.260s	4.060s	10m47.316s	45m9.069s
p13	0.380s	4.288s	11m46.216s	1m7.924s
p14	0.264s	3.616s	10m41.476s	37m33.689s
p15	0.220s	2.828s	7m53.570s	28m56.505s

Fig. 11. Experimental results

References

1. J.-R. Abrial. *The B-Book - Assigning programs to meanings*. Cambridge University Press, 2005.
2. J.A. Bergstra, A. Ponse, S.A. Smolka. *Handbook of Process Algebra*. Elsevier, 2001.
3. J.A. Bergstra, J. W. Klop. Algebra of Communicating Processes with Abstraction. *TCS*, 37:77–121, 1985.
4. T. Bolognesi, E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.
5. N. Brownlee. Failures-divergence refinement. Formal Systems (Europe) Ltd. In *Blount MetraTech Corp. Accounting Attributes and Record Formats* http://www.ietf.org/rfc/rfc2924.txt, 2000.
6. G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, F. Zulian. Specification and verification of the Powerscale^{TM} bus arbitration protocol: An industrial experiment with LOTOS. In *Proc. of FORTE*, 1996.
7. R. Chossart. Évaluation d'outils de vérification pour les spécifications de systèmes d'information. Master's thesis, Université de Sherbrooke, 2010.
8. 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.
9. E. M. Clarke, E. A. Emerson, A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
10. ClearSy. *Atelier B.* http://www.atelierb.societe.com.
11. 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.
12. R. De Nicola, F. Vaandrager. Action versus state based logics for transition systems. In *Proc. of the LITP spring school on theoretical computer science on Semantics of systems of concurrent processes*, Springer, 1990.

13. R. De Nicola, F. Vaandrager. Three logics for branching bisimulation (extended abstract). In *LICS*, pages 118–129, 1990.
14. M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, M. Ouenzar. Comparison of model checking tools for information systems. In *Proc. of ICFEM*, Springer, 2010.
15. M. Frappier, R. St.-Denis. EB 3: an entity-based black-box specification method for information systems. *Software and System Modeling*, 2003.
16. H. Garavel, F. Lang, R. Mateescu, W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *Proc. of TACAS*, Springer, 2011.
17. H. Garavel, G. Salaün, W. Serwe. On the semantics of communicating hardware processes and their translation into LOTOS for the verification of asynchronous circuits with CADP. *Science of Computer Programming*, 74(3):100–127, 2009.
18. F. Gervais. *Combinaison de spécifications formelles pour la modélisation des systèmes d'information*. PhD thesis, 2006.
19. C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.
20. D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
21. M. Leuschel, T. Massart, A. Currie. How to make FDR spin: LTL model checking of CSP by refinement. Technical report, 2000.
22. C. Löding, O. Serre. Propositional dynamic logic with recursive programs. In *Proc. of FOSSACS*, Springer, 2006.
23. J. Magee, J. Kramer. *Concurrency - State models and Java programs (2. ed.)*. Wiley, 2006.
24. R. Mateescu, G. Salaün. Translating Pi-calculus into LOTOS NT. In *Proc. of IFM*, Springer, 2010.
25. R. Mateescu, D. Thivolle. A model checking language for concurrent value-passing systems. In *Proc. of FM*, Springer, 2008.
26. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Inf. Comput.*, 100(1):1–40, 1992.
27. C. A. Petri. Concurrency. In *Advanced Course: Net Theory and Applications'75*, 1975.
28. A. W. Roscoe, Z. Wu. Verifying statemate statecharts using CSP and FDR. In *ICFEM*, 2006.
29. A.W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.
30. F. Lang, G. Salaün, R. Hérilier, J. Kramer, J. Magee. Translating FSP into LOTOS and networks of automata. *Formal Aspects of Computing*, 22(6):681–711, Springer, 2010.
31. F. Tronel, F. Lang, H. Garavel. Compositional verification using CADP of the Scalagent deployment protocol for software components. In *FMOODS*, Springer, 2003.

7 Appendix

$M_i^0\,(\overline{x}) = \|exp_i^{0,0}\,(\overline{x})\|$
$next\,(M_i)\,(\overline{x}) = \|exp_i^{j,k}\,(\overline{x})[f_j \leftarrow \text{if } j < i \text{ then } next\,(M_j) \text{ else } M_j]\|$,
where $1 \leq i \leq n,\ 1 \leq k \leq m_i$

$$R_{\mathsf{M}}-1 : \frac{}{(\rho, M) \xrightarrow{\rho} (\surd, next\,(M))}$$

$$R_{\mathsf{M}}-2 : \frac{(E_1, M) \xrightarrow{\rho} (E_1', M')}{(E_1.E_2, M) \xrightarrow{\rho} (E_1'.E_2, M')} \qquad R_{\mathsf{M}}-3 : \frac{(E, M) \xrightarrow{\rho} (E', M')}{(\surd.E, M) \xrightarrow{\rho} (E', M')}$$

$$R_{\mathsf{M}}-4 : \frac{(E_1, M) \xrightarrow{\rho} (E_1', M)}{(E_1|E_2, M) \xrightarrow{\rho} (E_1', M')} \qquad R_{\mathsf{M}}-5 : \frac{}{(E^*, M) \xrightarrow{\lambda} (\surd, M')}$$

$$R_{\mathsf{M}}-6 : \frac{(E, M) \xrightarrow{\rho} (E', M')}{(E^*, M) \xrightarrow{\rho} (E' \cdot E^*, M')} \qquad R_{\mathsf{M}}-7 : \frac{}{(\surd|[\Delta]|\surd, M) \xrightarrow{\lambda} (\surd, next\,(M))}$$

$$R_{\mathsf{M}}-8 : \frac{(E_1, M) \xrightarrow{\rho} (E_1', M')\ (E_2, M) \xrightarrow{\rho} (E_2', M')}{(E_1|[\Delta]|E_2, M) \xrightarrow{\rho} (E_1'|[\Delta]|E_2', M')} in\,(\rho, \Delta)$$

$$R_{\mathsf{M}}-9 : \frac{(E, M) \xrightarrow{\rho} (E', M')}{(GE \Rightarrow E, M) \xrightarrow{\rho} (E', M')}\ \|GE[f_i \leftarrow M_i]\|$$

$$R_{\mathsf{M}}-10 : \frac{(E_1, M) \xrightarrow{\rho} (E_1', M')}{(E_1|[\Delta]|E_2, M) \xrightarrow{\rho} (E_1'|[\Delta]|E_2, M')} \neg in\,(\rho, \Delta)$$

$$R_{\mathsf{M}}-11 : \frac{(E[\overline{x} := \overline{t}], M) \xrightarrow{\rho} (E', M')}{(P\,(\overline{t}), M) \xrightarrow{\rho} (E', M')} P\,(\overline{x}) = E$$

Fig. 12. EB^3 memory-based semantics Sem_{M}

```
module library is

type BOOKID is b1, b_bot with "eq", "ne", "ord" end type
type MEMBERID is m1, m2, m_bot with "eq", "ne", "ord" end type
type ACQUIR is array [0..1] of BOOL end type
type BOR is array [0..2] of MEMBERID end type

process book[ACQ, DIS: ANY](bid: BOOKID) is
var borrower: BOR in
  ACQ(bid); DIS(bid, ?borrower) where (borrower[ord(bid)] eq m_bot)
end var
end process

process loan[LEND, RET: ANY](mid: MEMBERID, bid: BOOKID) is
var acquired: ACQUIR, borrower: BOR in
  LEND(bid, mid, ?acquired, ?borrower)
     where ((borrower[ord(bid)] eq m_bot) and (acquired[ord(bid)] eq true)); RET(bid)
end var
end process

process member[REG, UNREG, LEND, RET: ANY](mid: MEMBERID) is
  REG(mid); loop L in select break L [] loan[LEND, RET](mid, b1) end select end loop;
  UNREG(mid)
end process

process memory[ACQ, DIS, REG, UNREG, LEND, RET: ANY] is
var mid: MEMBERID, bid: BOOKID, acquired: ACQUIR, borrower: BOR in
  borrower:= BOR(m_bot); acquired:= ACQUIR(false);
loop select
  ACQ(?bid); acquired[ord(bid)]:= true
[]
  DIS(?bid, ?borrower); acquired[ord(bid)]:= false
[]
  REG(?mid)
[]
  UNREG(?mid)
[]
  LEND(?bid, ?mid, !acquired, !borrower); borrower[ord(bid)]:= mid
[]
  RET(?bid); borrower[ord(bid)]:= m_bot
end select
end loop
end var
end process

process MAIN [ACQ, DIS, REG, UNREG, LEND, RET: ANY] () is

par ACQ, DIS, REG, UNREG, LEND, RET in
par
  loop L in select break L [] book[ACQ, DIS](b1) end select end loop
||
  par
    loop L in select break L [] member[REG, UNREG, LEND, RET](m1)
    end select end loop
  ||
    loop L in select break L [] member[REG, UNREG, LEND, RET](m2)
    end select end loop
  end par
end par
||
  memory[ACQ, DIS, REG, UNREG, LEND, RET]
end par
end process

end module
```

Fig. 13. LNT code for the Library Management System example (2 members, 1 book)

P1 is a classical liveness P. The second conjunct expresses the eventuality that a book be withdrawn from the library before it is reacquired.

```
macro P (B) =
  (
    (
      [ ( not { ACQ !B } )* ] < { ACQ !B } > true
    )
    and
    (
      [ true*. { DIS !B }. ( not { ACQ !B } )* ] < { ACQ !B } > true
    )
  )
end_macro

P ("B1") and P ("B2") and P ("B3")
```

P2 is a safety P.

```
macro P (B) =
  (
     [ true*. { ACQ !B }. ( not { DIS !B } )*. { ACQ !B } ] false
  )
end_macro

P ("B1") and P ("B2") and P ("B3")
```

P3.

```
macro P (B) =
  (
     (
        [ true*. ( ({ LEND ?ANY : STRING !B } or { TAKE ?ANY: STRING !B }) ).
          (not { RET !B })*. { DIS !B } ] false
     )
     and
     (
        [ true*. { RES ?ANY : STRING !B }.
          (not ( { CANCEL ?ANY : STRING !B } or { RET !B } ))*. { DIS !B } ] false
     )
  )
end_macro

P ("B1") and P ("B2") and P ("B3")
```

P4. The first conjunct 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 process *Lend* nor the reservation action *RES*.

```
macro P (M) =
  (
     (
        [ ( not { JOIN !M })*.
          ( {LEND !M ? ANY: STRING } or { TAKE !M ?ANY : STRING } ) ] false
     )
     and
     (
        [ true*. { LEAVE !M }.
          ( not { JOIN !M })*. ({LEND !M ? ANY: STRING } or { TAKE !M ?ANY : STRING } ) ] false
     )
  )
end_macro

P ("M1") and P ("M2")
```

Fig. 14. Verifications of requirements P1-P4

P5: 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 $RET\,!B$ 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. Notably the P:

```
[ (not ( { RES ?ANY: STRING !B } ))*. { RES ?ANY: STRING !B } ] false
```

is false as the regular expression inside the *box* modality may trigger a loan before the reservation.

```
macro P(B) =
  (
     [ ( not ( { LEND ?ANY: STRING !B } or { TAKE ?ANY: STRING !B } ))*.
       { RES ?ANY: STRING !B } ] false
  )
  and
  (
     [ true*. { RET !B }. ( not ( { LEND ?ANY: STRING !B } or { TAKE ?ANY: STRING !B } ))*.
       { RES !B } ] false
  )
  and
  (
     [ ( not ( { LEND ?ANY: STRING !B } or { TAKE ?ANY: STRING !B } or
               { TRANSFER ?ANY : STRING !B } or { RES ?ANY: STRING !B } ))*.
       { RES ?ANY: STRING !B } ] false
  )
end_macro

P("B1") and P ("B2") and P("B3")
```

P6: The difficulty here lies in the fact that the borrower may transfer the book to another member. For this reason, the specification

```
[ true*. { LEND !M !B }. (not ({ RET !B }))*. { RES !M !B } ] false
```

would be false as can be verified by the model checker. Note that we need the conjunction of all instances of the macro Q for possible values of books and members. Symmetry in the EB^3 specification may lead us to consider one instance only e.g. $Q\,(\text{"}B1\text{"},\text{"}M1\text{"})$ in the P thus reducing the time it needs to evaluate.

```
macro Q (B, M) =
(
   [    true*. { LEND !M !B }. (not ({ RET !B } or { TRANSFER ?M2: STRING !B } ))*.
        { RES !M !B } ]
   false
)
end_macro

Q("B1", "M1") and Q("B2", "M1") and Q("B3", "M1") and
Q("B1", "M2") and Q("B2", "M2") and Q("B3", "M2") and
Q("B1", "M3") and Q("B2", "M3") and Q("B3", "M3")
```

P7

```
macro Q (B, M) =
(
   [  true*. { RES !M !B }. (not ({ TAKE !M !B } or { CANCEL !M !B }))*.
      { RES !M !B } ]
   false
)
end_macro

Q("B1", "M1") and Q("B2", "M1") and Q("B3", "M1") and
Q("B1", "M2") and Q("B2", "M2") and Q("B3", "M2") and
Q("B1", "M3") and Q("B2", "M3") and Q("B3", "M3")
```

Fig. 15. Verifications of requirements P5-P7

P8: In this case, exploiting the symmetry is crucial to avoid the exponential state space explosion.

```
macro Q (B, M1, M2) =
(
   [ true*. { RES !M1 !B }. ( not ({ TAKE !M1 !B } or { CANCEL !M1 !B }) )*. { LEND !M2 !B } ]
   false
)
end_macro

Q("B1", "M1", "M2")
```

P9

```
macro Q (B, M) =
(
      [ true*. { RES !M !B  }. ( not ({ TAKE !M !B } or { CANCEL !M !B }) )*.
        { RENEW !B } ] false
)
end_macro

Q("B1", "M1")
```

P10: This P should be rephrased in the following way: It may not happen that a first member reserves a book and another member that reserves the book later takes it before the first member.

```
[
   true*.
   { RES ?M1: STRING ?B: STRING }.
   ( not ( { TAKE !M1 !B } or { CANCEL !M1 !B } or { TRANSFER !M1 !B } ))*.
   { RES ?M2: STRING !B where M2 <> M1 }.
   ( not ({ TAKE !M1 !B } or { CANCEL !M1 !B } or {TRANSFER !M1 !B } ))*.
   { TAKE !M2 !B }
] false
```

P11 corresponds to the classical P pattern: α_1 is not true between processes α_2 and α_3, which is expressed by the formula:
[true*. α_2. ($\neg$ α_3)*. α_1. ($\neg$ α_3)*. α_3] false, where
α_1 = ({ LEND !M !B } or { TAKE !M !B }) , α_2 = ({ LEND !M !B } or { TAKE !M !B }) and α_3 = { RET !B }.

```
macro Q (B, M) =
(
  [ true*. ({ LEND !M !B } or { TAKE !M !B }). ( not ( { RET !B } ) )*.
    ( { LEND !M !B } or { TAKE !M !B } ). ( not ( { RET !B } ))*.
    { RET !B } ] false
)
end_macro

Q("B1", "M1")
```

P12

```
macro Q (B, M) =
(
  [  true*. { RES !M !B }. ( not ({ TAKE !M !B } or { TRANSFER !M !B } ))* ]
     < ( not ( { TAKE !M !B } or { TRANSFER !M !B } ))*. { CANCEL !M !B } > true
)
end_macro

Q("B1", "M1")
```

Fig. 16. Verifications of requirements P8-P12

P13

```
macro Q (B, M) =
(
  (
     [ true*.
       ( {LEND !M !B } or { TAKE !M !B } ).
       ( not ( {RET !B } or { TRANSFER !"M2" !B } or { TRANSFER !"M3" !B } ))*.
       { LEAVE !M }. ( not ( {RET !B }  or { TRANSFER !"M2" !B } or { TRANSFER !"M3" !B } ))*.
       ( {RET !B }  or { TRANSFER !"M2" !B } or { TRANSFER !"M3" !B }) ] false
  )
  and
  (
     [ true*. { RES !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

Q("B1", "M1")
```

P14

```
macro Q (M) =
(
  [ true*. { JOIN !M }. ( not { LEAVE !M })* ] < ( not { LEAVE !M })*. { LEAVE !M } > true
)
end_macro

Q("M1")
```

P15: This P is dependent on the maximum number of books a member can have in his possession at any time. Supposing that this number is set to two the P can be written:

```
macro Q (M) =
(
  [ true*. let B1: STRING:= "B1", B2: STRING:= "B2" in
    ( { LEND !M !B1 } or { TAKE !M !B1 } ).
    ( not ({ TRANSFER ?M2: STRING !B1 } or { RET !B1 } ))*.
    ( { LEND !M !B2 } or { TAKE !M !B2 } ) end let ] false
)
end_macro

Q("M1")
```

Fig. 17. Verifications of requirements P13-15