Verification of EB3 specifications using CADP
Dimitris Vekris, Frédéric Lang, Catalin Dima, Radu Mateescu

To cite this version:
Verification of EB³ Specifications using CADP

Dimitris Vekris¹, Frédéric Lang², Catalin Dima¹, Radu Mateescu²

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. EB³ is a specification language for information systems. The core of the EB³ 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 EB³ specifications against temporal properties is of great interest to users of EB³. In this paper, we propose a translation from EB³ 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 EB³ and LNT specifications. We automated this translation with the EB³2LNT tool, thus equipping the EB³ method with the functional verification features available in the CADP toolbox.

Keywords: Process Algebras, Information Systems, EB³, LOTOS NT, Model Checking, Verification

1. Introduction

The EB³ method [FSî03] is an event-based paradigm tailored for information systems (ISs). EB³ has been used in the research projects SELKIS [MIL⁺11] and EB³SEC [JFG⁺10], whose primary aim is the formal specification of ISs with security policies. In the EB³SEC 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 EB³ specification defines entities, associations, and their respective attributes. The process algebraic nature of EB³ 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
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 $\mathsf{EB}^3$ must be completed with effective formal verification features.

Existing attempts for verifying $\mathsf{EB}^3$ specifications are based on translations from $\mathsf{EB}^3$ to other formal methods equipped with verification capabilities. A first line of work [GFL05, GFL06] focused on devising translations from $\mathsf{EB}^3$ attribute functions and processes to the B method [Abr05], which opened the way for proving invariant properties of $\mathsf{EB}^3$ 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 $\mathsf{EB}^3$ 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 $\mathsf{EB}^3$ to a target language accepted as input by a model checker was made so far.

At first sight, given that $\mathsf{EB}^3$ 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 [FS03] denoted $\mathsf{Sem}_T$ 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 [FS03] 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 $\mathsf{EB}^3$ 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 $\mathsf{Sem}_M$ 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, $\mathsf{Sem}_M$ are shown to be equivalent to $\mathsf{Sem}_T$. Again no complete translation of $\mathsf{EB}^3$ to LNT is provided.

**Contribution**

1. Based on the efficient memory $\mathsf{Sem}_M$ given in [VD13], we propose a rigorous translation algorithm from $\mathsf{EB}^3$ to LNT [CCG05]. As far as we know, this is the first attempt to provide a general translation from $\mathsf{EB}^3$ to a classical value-passing process algebra. Since our primary objective was to provide temporal property verification features for $\mathsf{EB}^3$, 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 $\mathsf{EB}^3$ 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 $\mathsf{EB}^3$ guarded expressions containing attribute function calls. At last, the constructed data types and pattern-matching mechanisms of LNT enabled a natural description of $\mathsf{EB}^3$ data types and attribute functions.

2. We implemented our translation in the $\mathsf{EB}^3\text{-}\text{LNT}$ tool, thus making possible the analysis of $\mathsf{EB}^3$ 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 EB³ 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 EB³ specification that serves as the basis for introducing EB³’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 $\text{Sem}_M$ of EB³ and compares it in terms of efficiency to the standard traced-based semantics $\text{Sem}_T$.
- Section 7 gives an overview of the EB³ and LNT languages.
- Section 8 introduces MCL briefly.
- Section 9 presents our translation from EB³ to LNT, implemented by the EB³2LNT translator.
- Section 10 shows how EB³2LNT, 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 EB³ 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 EB³ specification intended to satisfy these requirements is presented in Figure 1. Note that process $\text{main}$ is the parallel interleaving between $m$ instances of process $\text{book}$ executed an arbitrary but bounded number of times (Kleene Closure $^*$) and $p$ instances of process $\text{member}$ executed an arbitrary but bounded number of times. Process $\text{book}$ stands for a book acquisition “$\text{Acquire (bId)}$” followed by its eventual discard “$\text{Discard (bId)}$”. The attribute function “$\text{borrower (T, bId)}$” looks for actions of the form “$\text{Lend (mId, bId)}$” or “$\text{Return (bId)}$” in the trace $T$ and returns the current borrower of book $\text{bId}$ denoted as “$\text{borrower (front (T), bId)}$” or $\bot$ if the book is not lent. In process $\text{book}$, action “$\text{Discard (bId)}$” is thus guarded to guarantee that book $\text{bId}$ cannot be discarded if it is currently lent. Process $\text{loan}$ stands for a book loan “$\text{Lend (bId, mId)}$” followed by an eventual return to the library “$\text{Return (bId)}$”. The attribute function “$\text{nbLoans (T, mId)}$” inspects for actions of the form “$\text{Lend (bId, mId)}$” incrementing the current number of $\text{mId}$’s loans and inspects for actions of the form “$\text{Return (bId)}$” decrementing the current number of $\text{mId}$’s loans. In process $\text{loan}$, action “$\text{Lend (bId, mId)}$” is thus guarded to guarantee that book $\text{bId}$ is not currently being lent and member $\text{mId}$’s loans is inferior to $\text{NbLoans}$. Process $\text{member}$ stands for a member registration followed by an arbitrary but bounded number of loans and an eventual membership annulment.

3. The Language EB³

The EB³ method has been specially designed to specify the functional behaviour of ISs. A standard EB³ specification comprises:

1. a class diagram representing entity types and associations for the IS being specified,
2. a process algebra specification, denoted by $\text{main}$, 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.
We limit the presentation to the process algebra and the set of attribute functions. The attribute function definitions are denoted by the symbol $\alpha$. The trace without its last action is denoted by $T_{\bot}$. The symbol $T \ast$ denotes the last action of the trace, and $front(T)$ which denotes the trace without its last action. The symbol $\bot$ represents the undefined value.

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 $\rho$ is either the internal action written $\lambda$, or a visible action of the form “$\alpha(\bar{\tau})$”, 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., $\bar{\tau}$ 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 “$P(\bar{\tau}) = E^\prime$”, 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 $Lab = \{\alpha_1, \ldots, \alpha_q\}$ ($\lambda$ not included). Each $\alpha_i$ has formal parameters $\bar{\tau}_i$. Attribute functions $\{f_1, \ldots, f_n\}$ are defined recursively on the current trace $\bar{T}$ representing the history of actions with the aid of functions $last(T)$ which denotes the last action of the trace, and $front(T)$ which denotes the trace without its last action. The symbol $\bot$ represents the undefined value. In particular, both

![Fig. 1. $EB^3$ specification of a library management system](image)

![Fig. 2. $EB^3$ syntax](image)
Expressions are built over variables, constants, and standard operations. An action \( \rho \) is an expression of the same type as \( f \)'s return type built over the variables \( \overline{\gamma} \cup \overline{\pi} \). In real EB\(^3\) specifications with distinct formal parameters \( \overline{\gamma} \), each corresponding to every \( f_i \), \( \overline{\gamma} \) can be formalized as the union of the \( \overline{\gamma}_i \), \( i \in 1..n \). 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 \( "f_h (\text{front} (T), ...)" \) occurring in \( w_i' \) satisfies \( h < i \) and every call of the form \( "f_h (\text{front} (T), ...)" \) 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 \textit{attribute function} evaluation. In particular, the definition of an attribute function \( f_i \) cannot contain recursive calls of the form \( "f_i (\text{front} (T), \ldots)" \), but only recursive calls of the form \( "f_i (\text{front} (T), \ldots)" \). Note that this does not limit the expressiveness of EB\(^3\) \textit{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 \textit{borrower} contains calls of the form \( "\text{borrower} (\text{front} (T), bId)" \), while the definition of \textit{nbLoans} contains calls of the form \( "\text{nbLoans} (\text{front} (T), mId)" \) and \( "\text{borrower} (T, bId)" \).

**Process expressions.** Expressions are built over variables, constants, and standard operations. An action \( \rho \) is the simplest process expression, whose semantics are given by Rule M1. The symbol \( \sqrt{\rho} \) (which is not part of the user syntax) denotes successful execution. The \textit{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 \( \lambda \) does not appear in the trace.) At system start, the trace is empty. If \( T \) denotes the current trace and action \( \rho \) can be executed, then \( T.\rho \) denotes the trace just after executing \( \rho \).

EB\(^3\) processes can be combined with classical process algebra operators such as the \textit{sequence} \( "E_1, E_2" \) (M2, M3), the \textit{choice} \( "E_1 | E_2" \) (M4) and the \textit{Kleene closure} \( "E_0^*" \) (M5, M6). Rules (M7 to M9) define \textit{parallel} composition \( "E_1 [\Delta] E_2" \) of \( E_1, E_2 \) with synchronization on \( \Delta \subseteq \text{Lab} \). The condition \( "\text{in} (\rho, \Delta)" \) is true iff the label of \( \rho \) belongs to \( \Delta \). 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 \( "E_1 [\text{Lab}] E_2" \).

\(^1\) each \( y \in \overline{\gamma} \) is called \textit{attribute parameter} correspondingly
Quantification is permitted for choice and parallel composition. If $V$ is a set of expressions $\{v_1, \ldots, v_n\}$, 
"$\forall x \in V : E_0$" and "$(\forall \Delta) \exists x \in V : E_0$" stand respectively for "$E_0[x := v_1] \land \ldots \land E_0[x := v_n]$" and "$E_0[x := v_1] \land (\forall \Delta) \exists E_0[x := v_n]$", where "$E[x := v]$" denotes the replacement of all occurrences of $x$ by $v$ in $E$. For instance, "$(\forall x \in \{1, 2, 3\}; a(x)$" stands for "$a(1) \land a(2) \land a(3)$".

At last, named processes can be instantiated as usual $(M_{i1})$. 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 certainty 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 $\phi$ in question is expressed in Linear Tree Logic (LTL) [Pnu77]. Then, the Büchi automaton corresponding to $\phi$ 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 $\phi$. In [Gro06], the liveness property $\phi$ is expressed as proof obligations of the Event-B model, whose resolution implies the satisfaction of $\phi$. 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 $\psi$ is valid, there is a sequence of transitions that lead to states of the system, where formula $\phi$ 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 trigger 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 [LMC06]. 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,
The memory after its execution. By definition, the inert action is successively updated from the original trace-based memory $\lambda$. Calls of the form $f_i[h]$ upon execution of $f_i$ where the notation $\parallel$ 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 $\alpha_j(\overline{\sigma})$ can be executed, then $M'$ denotes the memory after its execution. By definition, the inert action $\lambda$ has no effect on the memory $M$, i.e. $M' = M$. The memory $M$ is successively updated from $f_1,..,f_n$ to $M' = \{f'_1,..,f'_n\}$ as depicted in Definition (3). Calls of the form $f_h(T,\overline{\sigma})$ within $w'_j$ are replaced by already-updated $f_h[\overline{\sigma}]$, since $f_h$ takes $T$ as argument for $h < i$. Calls of the form $f_h(T,\overline{\sigma})$ are replaced by the value of $f_h[\overline{\sigma}]$, since $f_h$ takes $front(T)$ as argument for $h \geq 0$. This computation caters for the complete discharge of the trace $T$ from $w'_j$, while consistently

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 $Sem_T$ [FS03] in terms of efficiency. For defining the $Sem_M$ semantics for attribute functions, attribute functions are turned into state variables (the memory $M = \{f_1,..,f_n\}$) 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 $f_i(T, x_1 : T_1,..,x_l : T_l)$ is an attribute function, we construct $|T_1| \times \ldots \times |T_l|$ state variables denoted by $h$, where $|T_i| \{i \in 1..l\}$ stands for $T_i$’s cardinality. Then, $f_i^h$ denotes $f_i$’s update upon execution of $\alpha_j(\overline{\sigma})$ is given as follows:

\[
M' = \{f'_1,..,f'_n\} \tag{1}
\]

\[
f^h \parallel w^j \parallel f_h(T,\overline{\sigma}) \leftarrow f^h[\overline{\sigma}] \mid h < i \] \tag{2}
\]

\[
[ f_h(front(T),\overline{\sigma}) \leftarrow f_h[\overline{\sigma}] \mid h \geq 0 ] \parallel (\overline{\sigma}) \] \tag{3}
\]

where the notation $\parallel ||$ 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 $\alpha_j(\overline{\sigma})$ can be executed, then $M'$ denotes the memory after its execution. By definition, the inert action $\lambda$ has no effect on the memory $M$, i.e. $M' = M$. The memory $M$ is successively updated from $f_1,..,f_n$ to $M' = \{f'_1,..,f'_n\}$ as depicted in Definition (3). Calls of the form $f_h(T,\overline{\sigma})$ within $w^j$ are replaced by already-updated $f_h[\overline{\sigma}]$, since $f_h$ takes $T$ as argument for $h < i$. Calls of the form $f_h(T,\overline{\sigma})$ are replaced by the value of $f_h[\overline{\sigma}]$, since $f_h$ takes $front(T)$ as argument for $h \geq 0$. This computation caters for the complete discharge of the trace $T$ from $w^j$, while consistently
Memory

<table>
<thead>
<tr>
<th>State</th>
<th>borrower([b_1])</th>
<th>borrower([b_2])</th>
<th>nbLoans([m_1])</th>
<th>nbLoans([m_2])</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>A'</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>B</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>B'</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>⊥</td>
<td>⊥</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>(m_1)</td>
<td>⊥</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 4. Evolution of Attribute variables

updating the memory \(M\). We denote by \(M_0 = \{f_0^0, \ldots, f_n^0\}\) the memory at system start, which corresponds to “\(T = \emptyset\)” and “\(i = 0\)” and set \(f_i^0[\overline{v}] = \perp\) for every parameter vector \(\overline{v}\) 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 “[\(C \Rightarrow f_i(T, \overline{v}) := f_i[\overline{v}]\)]” such that “\(M = \{f_1, \ldots, f_n\}\)”.

Since \(C\) may contain calls to attribute functions of the form \(f(T, \overline{v})\), the evaluation of \(C\) necessitates their replacement by their corresponding attribute variables \(f[\overline{v}]\). 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 \(\rho\) 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 \(E^3\) 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:

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

The first two cells keep the two values of the attribute function \(\text{borrower}(T, \bullet)\) for a given trace \(T\), and the last two keep the values of \(\text{nbLoans}(T, \bullet)\). 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 \((\perp, \perp, \perp, \perp)\) After the trace “\(\text{Acquire}(b_1)\).\(\text{Acquire}(b_2).\text{Register}(m_1).\text{Register}(m_2)\)” the memory contains \((\perp, \perp, \perp, 0)\). If the action “\(\text{Lend}(b_1, m_1)\)” is then executed, the new memory is \((m_1, \perp, 1, 0)\). For instance, the new value \(m_1\) for \(\text{borrower}[b_1]\) is obtained from the Rule “\(\text{Lend}(b_1, m_1)\) : \(\text{nbLoans}(\text{front}(T), \text{mId}) + 1\)” of the attribute function \(\text{nbLoans}\), where the value of \(\text{nbLoans}(\text{front}(T), m_1)\) corresponds to the value of \(\text{nbLoans}[m_1]\) 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 \(\text{Acquire}(b_1)\), \((B)\) is the state the moment succeeding the execution of \(\text{Acquire}(b_2)\), \((B')\) is the state the moment succeeding the execution of \(\text{Register}(m_1)\), \((C)\) is the state the moment succeeding the execution of \(\text{Register}(m_2)\) and \((D)\) is the state the moment succeeding the execution of \(\text{Lend}(b_1, m_1)\).

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

\[
(borrower(T,b_1) = \perp) \land (nbLoans(T,m_1) < 2)
\]  

(4)

where \(T\) corresponds to the system trace at state \((C)\), i.e. \(T = T_C\). 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 “\(\text{borrower}(T,b_1)\)” w.r.t. \(Sem_M\) and then we compare it with the evaluation of “\(\text{borrower}(T,b_1)\)” w.r.t. the
main  (A)

\[ \text{Acquire} (b_2). \text{Acquire} (b_1) \rightarrow \]

\[ \text{borrower} (T_B, b_1) = \perp \rightarrow \text{Discard} (b_1). \text{book} (b_1)^* \]
\[ \| \]

\[ \text{borrower} (T_B, b_2) = \perp \rightarrow \text{Discard} (b_2). \text{book} (b_2)^* \]
\[ \| \]

\[ ( \| MId : MID : \text{member} (mId)^* ) \quad (B) \]

\[ \text{Register} (m_1). \text{Register} (m_2) \rightarrow \]

\[ \text{borrower} (T_C, b_1) = \perp \rightarrow \text{Discard} (b_1). \text{book} (b_1)^* \]
\[ \| \]

\[ \text{borrower} (T_C, b_2) = \perp \rightarrow \text{Discard} (b_2). \text{book} (b_2)^* \]
\[ \| \]

\[ ( \| bId : BID : \text{loan} (m_1, bId)^* ). \text{Unregister} (m_1). \text{member} (m_1)^* \]
\[ \| \]

\[ ( \| bId : BID : \text{loan} (m_2, bId)^* ). \text{Unregister} (m_1). \text{member} (m_2)^* \quad (C) \]

\[ \text{Lend} (b_1, m_1) \rightarrow \]

\[ \text{borrower} (T_D, b_1) = \perp \rightarrow \text{Discard} (b_1). \text{book} (b_1)^* \]
\[ \| \]

\[ \text{borrower} (T_D, b_2) = \perp \rightarrow \text{Discard} (b_2). \text{book} (b_2)^* \]
\[ \| \]

\[ (\text{Return} (b_1). \text{loan} (m_1, b_1)^* \| \text{loan} (m_1, b_2)^*). \text{Unregister} (m_1). \text{member} (m_1)^* \]
\[ \| \]

\[ ( \| bId : BID : \text{loan} (m_2, bId)^* ). \text{Unregister} (m_1). \text{member} (m_2)^* \quad (D) \]

Fig. 5. Sample execution

classical traced-based memory \( Sem_T \). Following (3) and the definition of \( \text{borrower} \) in Figure 1, attribute function call “\( \text{borrower} (T, b_1) \)” is evaluated w.r.t. \( Sem_M \) as follows:

\[ \| \text{borrower} (T_C, b_1) \| = \text{borrower}_B [b_1] = \perp \]

where “\( \text{borrower}_B [b_1] \)” is the (stocked) attribute variable at state (B’). Following the attribute function definition of \( \text{borrower} \) in Figure 1, the evaluation of “\( \text{borrower} (T_C, b_1) \)” w.r.t. \( Sem_T \) (the standard ML-like evaluation of recursive functions denoted as \( \| . \|_T \) to mark the difference with the \( \| . \| \) evaluation) triggers the complete traversal of the whole trace as demonstrated below:

\[ \| \text{borrower} (T_C, b_1) \|_T = \| \text{borrower} (\text{front} (T_C), b_1) \|_T \]
\[ = \| \text{borrower} (T_{B'}, b_1) \|_T \]
\[ = \| \text{borrower} (\text{front} (T_{B'}), b_1) \|_T \]
\[ = \| \text{borrower} (T_B, b_1) \|_T \]
\[ = \| \text{borrower} (\text{front} (T_B), b_1) \|_T \]
\[ = \| \text{borrower} (T_A, b_1) \|_T \]
\[ = \| \text{borrower} (\text{front} (T_A), b_1) \|_T \]
\[ = \| \text{borrower} (T_A, b_1) \|_T \]
\[ = \| \perp \|_T = \perp \]

from which follows directly that \( Sem_M \) is computationally more efficient than \( Sem_T \).
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 (\(\text{select}\)), conditional (\(\text{if}\)), sequential composition (\(;\)), breakable loop (\(\text{loop}\) and \(\text{break}\)) and parallel composition (\(\text{par}\)). Communication is carried out by rendezvous on gates, written \(G, G_1, \ldots, G_m\), and may be guarded using Boolean conditions on the received values (\(\text{where}\) 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 “\(\text{break}\) \(L\)” 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 \(\mu\)-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 \(\alpha\) and state formulas \(\phi\). Let a set of data variables \(X\) and a set of function identifiers \(F\) with standard interpretation. Data expressions \(e\) are defined as follows:

\[
e := x \mid f(e_1, \ldots, e_n),
\]

where \(x \in X\) and \(f \in F\) denoting typed function identifiers.

Action formulas \(\alpha\) consist of:

- action patterns of the form \(\{ e \mid e_1 \ldots \mid e_n \}\) such that, if action \(\{ e \mid v_1 \ldots \mid v_n \}\) occurs, \(v_k\) matches expression \(e_k\) for \(k \in \{1, \ldots, n\}\),
- action patterns of the form \(\{ e \not\! x_1 : T_1 \ldots \not\! x_n : T_n \}\) such that, if action \(\{ e \mid v_1 \ldots \mid v_n \}\) occurs, \(v_k\) is assigned to variable \(x_k\) for \(k \in \{1, \ldots, n\}\), and
- usual Boolean operators.

To sum up, the abstract syntax of action formulas \(\alpha\) is given by the following grammar:

\[
\alpha := \{ e \mid e_1 \ldots \mid e_n \} \mid \{ e \not\! x_1 : T_1 \ldots \not\! x_n : T_n \} \mid \neg \alpha \mid \alpha_1 \land \alpha_2 \mid \alpha^*.
\]

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

\(T_k\) denotes \(x_k\)’s type as well as the corresponding domain of elements that can be assigned to \(x_k\).

---

\[\begin{align*}
B & ::= \text{stop} \mid \text{null} \mid G(O_1, \ldots, O_n) \text{ where } E \mid B_1 \mid B_2 \\
& \quad \mid \text{if } E \text{ then } B_1 \text{ else } B_2 \text{ end if } \mid \text{var } x : T \text{ in } B \text{ end var } \mid x := E \\
& \quad \mid \text{loop } L \text{ in } B \text{ end loop } \mid \text{break } L \mid \text{select } B_1 \mid \ldots \mid B_n \text{ end select} \\
& \quad \mid \text{par } G_1, \ldots, G_m \text{ in } B_1 \mid \ldots \mid B_n \text{ end par } \mid P[G_1, \ldots, G_m](E_1, \ldots, E_n) \\
O & ::= \! E \mid \? x
\end{align*}\]

Fig. 6. LNT syntax (limited to the fragment used in this paper)
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 [EL96].

We assign values to propositional variables $x \in X$ appearing in expressions $e$ and action formulas $a$ that are denoted as $\delta : X \rightarrow T_1 \cup \ldots \cup T_n$. Moreover, we assign functions to all free propositional variables $Y$ appearing in MCL state formulas that are denoted as $\rho : Y \rightarrow F$. 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 wildcard 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:

\[
\phi ::= \ e \ | \ \neg \phi \ | \ \phi_1 \lor \phi_2 \ | \ \langle \alpha \rangle \phi \ | \ \mu Y. \phi \\
| \ \exists x_1 : T_1, \ldots, x_n : T_n. \phi \ | \ Y(e_1, \ldots, e_n) \\
| \ \text{let } x_1 : T_1 := e_1, \ldots, x_n := e_n \text{ in } \phi \ \text{end let} \\
| \ \text{if } \phi_1 \ \text{then } \phi_2 \ \text{elsif } \phi_3 \ \text{then } \phi_4 \ \text{else } \phi_5 \ \text{end if} \\
| \ \text{case } e \ \text{is } p_1 \rightarrow \phi_1 \ | \ \ldots \ | \ p_n \rightarrow \phi_n \ \text{end case}
\]

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

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

Let’s consider the following MCL example:

\[
[\text{true}^* . \{ \text{RESERVE } ?M : \text{string} \ ; ?B : \text{string} \} . \{ \text{not } (\{ \text{TAKE } !M \ ; !B \})^* . \{ \text{RESERVE } !M \ ; !B \} \} ] \text{ false}
\]

The expression inside $[\ldots]$ describes the set of execution paths starting with a bounded number of actions (denoted as 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 $[\ldots]$ 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 $f_h(T, \ldots)$ (the value of $f_h$ on the current trace), whereas calls of the form $f_h(front(T), \ldots)$ are replaced by accesses to a copy $\overline{f}$ of the memory $\overline{T}$, 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 $a_j$ of the system. At this point, we recall the existence of unique attribute function definition $f_i$ ($T : T, y_1 : T_1, \ldots, y_n : T_n) : T$ (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 $\overline{f}$, so that:

\[f_i[\text{ord}(v_1)] \ldots [\text{ord}(v_n)]\]
process $M \{ \alpha_1, \ldots, \alpha_q, \lambda : \text{any} \}$ is
\[
\begin{align*}
\text{var } \vec{f}, \vec{g} : T, & \quad \vec{y} : T_{at}, \quad \vec{x} : T_{ac} \text{ in} \\
\text{upd}^0; & \quad \ldots; \quad \text{upd}^n;
\text{loop} \\
\vec{y} := \vec{f} & \\
\text{select} \\
\alpha_1(?\vec{x},!\vec{f}) & \quad \text{upd}_1; \quad \ldots; \quad \text{upd}_n \\
\emptyset & \quad \ldots \quad \emptyset \\
\alpha_q(?\vec{x},!\vec{f}) & \quad \text{upd}_q; \quad \ldots; \quad \text{upd}_n \\
\emptyset & \quad \lambda(!\vec{f}) \\
\text{end select} \\
\text{end loop} \\
\text{end var} \\
\end{align*}
\]
end process

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

\[
\begin{align*}
\text{upd}_i^j & \doteq \text{enum} (\vec{g}, f_i [\text{ord} (\vec{y})] := \text{mod} (w_i^j)) \\
\text{enum} (\emptyset, B) & \doteq B \\
\text{enum} (y_k :: \vec{x}, B) & \doteq y_k := \text{first}_k; \\
\text{loop } & \text{L}_k \text{ in} \\
\text{enum} (\vec{x}, B) & \\
\text{if } y_k \neq \text{last}_k \text{ then } y_k := \text{next}_k(y_k) \text{ else break } L_k \text{ end if} \\
\text{end loop} \\
\text{mod} (w_i^0) & \doteq w_i^0 \quad [ f_h (T, \vec{y}) := f_h [\text{ord} (\vec{y})] \quad \forall h < n ] \\
& \quad [ f_h (\text{front}(T), \vec{y}) := \bot \quad \forall h \in \{1, \ldots, n\} ] \\
\text{mod} (w_i^j) & \doteq w_i^j \quad [ f_h (T, \vec{y}) := f_h [\text{ord} (\vec{y})] \quad \forall h < n ] \\
& \quad [ f_h (\text{front}(T), \vec{y}) := g_h [\text{ord} (\vec{y})] \quad \forall h \in \{1, \ldots, n\} ], \text{if } j \neq 0
\end{align*}
\]

Fig. 8. LNT code for expression $\text{upd}_i^j$ of Figure 7

encodes the current value of:
\[ f_i (T, v_1, \ldots, v_n), \]
where $\text{ord}(v_k)$ 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 $\text{upd}_i^j$ for $i \in \{1, \ldots, q\}$ and $j \in \{1, \ldots, q\}$ of Figure 7 is used to implement the effect of action $\alpha_i(\vec{x})$ on attribute variables $f_i$. Its definition is given in Figure 8. In particular, $\text{upd}_i^j$ is defined by way of auxiliary function $\text{enum}$. For vector $\vec{x} = (z_1, \ldots, z_n)$, $y_k :: \vec{x}$ is equivalent notation for $(y_k, z_1, \ldots, z_n)$ and $[\ ]$ stands for the empty vector. For each type $T_k$, we assume the existence of functions $\text{first}_k$ that returns the first element of type $T$, $\text{last}_k$ that returns the last element of type $T_k$, and $\text{next}_k(x)$ that returns the successor of $x$ on condition that the type of variable $x$ is $T_k$ (following the total order induced by $\text{ord}$). For example, let $T_k = \{m_1, m_2, m_3\}$. Then, it is $\text{first}_k = m_1$, $\text{next}_k(m_1) = m_2$, $\text{next}_k(m_2) = m_3$ and $\text{last}_k = m_3$. According to the definition of “$\text{enum} (y_k :: \vec{x}, B)$” of Figure 8, the $\text{loop}$ 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 \textit{enum} \((\pi, B)\) is taken. If \(y_k\) has not been assigned the last value of \(D_k\), which is expressed by condition \(y_k \neq \text{\textit{last}}_s\), the next value of \(D_k\) is assigned to \(y_k\) via \(y_k := \text{\textit{next}}_s(y_k)\) and \textit{enum} \((\pi, B)\) is called once again. Otherwise, the program breaks from the \textbf{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 \textit{enum} depends on function \textit{mod} \((E)\) which transforms an expression \(E\) by syntactically replacing function calls by array accesses as described previously. Namely according to the definition of \textit{mod} \((w^j)\) of Figure 8, expressions of the form \("f_h(T, \pi)"\) are replaced by \(f_h[\text{\textit{ord}}(\pi)]\) and expressions of the form \("f_h(f_r(T), \pi)"\) are replaced by \(g_k[\text{\textit{ord}}(\pi)]\) for all \(h < n\). Recall at this point that the initial values of attribute variables \(\pi\) are by convection equal to \(\bot\), which entails the replacement of attribute function calls of the form \("f_h(f_r(T), \pi)"\) in \(\mathbb{EB}\) type (1) value expressions \(w^0\) by \(\bot\) for all \(h \in \{1, \ldots, n\}\). Notation \(\pi : T_{at}\) is an abbreviation for \("y_1 : T_1, \ldots, y_s : T_s"\) and \(T\) denotes the vector \((T, \ldots, T)\) of size \(n\) that is equal to the number of attribute function names in the system.

Expression \textit{next}_s \((x)\) is implemented as LNT expression \(\text{\textit{val}}(\text{\textit{ord}}(x)+1)\). Note that in LNT assignment \("x := \text{\textit{val}}(i)"\) identifier \text{\textit{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 \(\pi := (x_1, \ldots, x_p)\) for all action labels \(\alpha_j\) 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 \(\pi : T_{ac}\) 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 \(y_k \in \pi\) for \(k \in \{1, \ldots, s\}\) as seen earlier. Then, upon synchronization on action \(\alpha_j(\pi, T)\) with the LNT process corresponding to \(\mathbb{EB}\)'s \textit{main} process (see translation of processes below), the values of all attribute variables \(\alpha_i\) for \(i \in \{1, \ldots, n\}\) are updated using function \textit{up}_\(\pi\).

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

\begin{aligned}
\text{\textit{type}}\ MId & \text{\textit{is}} m_1, m_2, m_\bot \text{ with } \"eq\", \"ne\", \"ord\", \"val\" \text{\textit{end type}} \\
\text{\textit{type}}\ BId & \text{\textit{is}} b_1, b_\bot \text{ with } \"eq\", \"ne\", \"ord\", \"val\" \text{\textit{end type}}
\end{aligned}

Identifier \textit{eq} (\textit{ne}) denotes the equality (inequality) operator among member IDs or book IDs. The bottom value for member IDs is denoted as \(m_\bot\) and the bottom value for book IDs is denoted as \(b_\bot\). 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:

\begin{aligned}
\text{\textit{type}}\ NB & \text{\textit{is}} \text{array}[0..2] \text{ of } NAT \text{\textit{end type}} \\
\text{\textit{type}}\ BOR & \text{\textit{is}} \text{array}[0..1] \text{ of } MId \text{\textit{end type}}
\end{aligned}

Now, we need to explain how the execution of communication label \textit{LEND} modifies attribute variable vectors \("borrower : BOR\) and "\textit{nbLoans} : \textit{NB}". First, we remark that, according to the \(\mathbb{EB}\) specification of the library management system, the execution of \(\mathbb{EB}\) action \("\textit{Lend} (bId, mId)"\) modifies attribute variables \textit{borrower}[\textit{bId}] and \textit{nbLoans}[\textit{mId}]. Hence, following the definition of Figure 7, we need an auxiliary variable \(\textit{bId}'\) 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 \textit{borrower}[\textit{bId}] in case that \textit{book} \textit{bId} is lent to member \textit{mId}. In each iteration of the \textbf{loop} construct, it is checked if \textit{bId} is equal to the current value of \textit{bId}', in which case \textit{borrower}[\textit{bId}] is set to the current borrower, i.e. \textit{mId}. Then, it is checked if \textit{bId}' is equal to the last element of \(BID\)'s domain, i.e. \(b_1\), in which case the program control breaks from the \textbf{loop} construct. Otherwise, \textit{bId}' is assigned the next element in the ordered set of \(BID\)'s domain, i.e. \("bId' := \text{\textit{val}}(\text{\textit{ord}}(bId') + 1)"\), and the execution of the \textbf{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 \textit{nbLoans}[\textit{mId}] in case that \textit{book} \textit{bId} is lent to member \textit{mId}. In each iteration of the \textbf{loop} construct, it is checked if \textit{mId} coincides with the current value of \textit{mId}', in which case \textit{nbLoans}[\textit{mId}] is increased by one. Then, it is checked if \textit{mId}' is equal to the last element of \(MID\)'s domain, i.e. \(m_2\), in which case the program control breaks from the \textbf{loop} construct. Otherwise, \textit{mId}' is assigned the next element in the ordered set of \(MID\)'s domain, i.e. \("mId' := \text{\textit{val}}(\text{\textit{ord}}(mId') + 1)"\), and the execution of the \textbf{loop} construct is repeated.
The code for the memory $M$ is then given as follows:

```plaintext
process $M$ [Acquire, Discard, Register, Unregister, Lend, Return : any] is
  var $mId : MID$, $bId : BID$, $mId' : MID$, $bId' : BID$, $borrower : BOR$, nbLoans : NB in
  borrowee := BOR(m_⊥); nbLoans := NB(0);
loop
  select
    Acquire (?bId)
    Discard (?bId, ?borrower)
    Register (?mId)
    Unregister (?mId)
  Lend (?bId, ?mId, nbLoans, ?borrower);
  $bId' := b_1$;
loop $L_1$ in
    if ($bId' eq bId$) then
      borrowee [ord ($bId'$)] := $mId$
    end if;
    if ($bId' eq b_1$) then
      break $L_1$
    else
      $bId' := val (ord ($bId'$) + 1)$
    end if;
end loop
$mId' := m_1$;
loop $L_2$ in
  if ($mId' eq mId$) then
    nbLoans [ord ($mId'$)] := nbLoans [ord ($mId'$)] + 1
  end if;
  if ($mId' eq m_2$) then
    break $L_2$
  else
    $mId' := val (ord ($mId'$) + 1)$
  end if
end loop
RET (?bId); $mId' := m_1$;
loop $L_1$ in
  if ($mId' eq borrowee [ord ($bId$)]$) then
    nbLoans [ord ($mId'$)] := nbLoans [ord ($mId'$)] − 1
  end if;
  if ($mId' eq m_2$) then
    break $L_1$
  else
    $mId' := val (ord ($mId'$) + 1)$
  end if
end loop
```
\[ bId' := b_1; \]
\[
\text{loop } L_1 \text{ in }
\]
\[
\text{if } (bId' \text{ eq } bId) \text{ then }
\]
\[
\centeqn{\text{borrower [ord (bId)] := } \bot} \]
\[
\text{end if; }
\]
\[
\text{if } (bId' \text{ eq } b_1) \text{ then }
\]
\[
\text{break } L_1
\]
\[
\text{else }
\]
\[
bId' := \text{val (ord (bId') + 1)}
\]
\[
\text{end if}
\]
\[
\text{end loop; }
\]
\[
\text{end select }
\]
\[
\text{end loop }
\]
\[
\text{end var }
\]
\[
\text{end process }
\]

Note that LNT statement “\(\text{borrower := BOR(\bot)}\)” is equivalent to “\(\text{borrower[ord (b_1)] := } \bot\)”. Similarly, “\(\text{nbLoans := NB(0)}\)” is equivalent to “\(\text{nbLoans[ord (m_1)] := 0}; \text{ nbLoans[ord (m_2)] := 0}\)”. Note that the initial number of loans is set to 0, whereas according to Figure 1 it should be set to \(\bot\). The reason is that it is impossible to define \(\text{Nat}_\bot\) of Figure 1 without recourse to complex LNT data structures. Hence, the symbol \(\bot\) 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 \(\lambda\) has been removed from the previous LNT specification, since \(\lambda\) 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 \(\mathcal{F} = (\text{nbLoans, borrower})\) is removed from the parameter vector of communication labels that synchronize with expressions of the form “\(\alpha (\tau, \mathcal{F}) \text{ where } \text{mod (C)}\)” present in other LNT processes (see Figure 9), for which the corresponding guard \(\text{mod (C)}\) makes no use of attribute variable vector \(\mathcal{F}\). In particular, this optimization applies to communication labels \(\text{Acquire, Register, Unregister} \text{ and Return}\). 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 \(\mathcal{F} = (\text{nbLoans, borrower})\) passed as parameters to the communication labels “\(\alpha (\tau, \mathcal{F}) \text{ where } \text{mod (C)}\)” are exactly those appearing in \(\text{mod (C)}\). Notice, for example, that vector \(\text{nbLoans}\) is not a subset of the parameter vector passed to communication label \(\text{Discard}\), since the static analysis of the LNT specification reveals that no corresponding guard to \(\text{Discard}\) makes use of \(\text{nbLoans}\).

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

It turns out that the code simulating the effect of \(\text{Eb3}^3\) action “\(\text{Lend (bId, mId)}\)” on attribute variables \(\text{borrower[bId]}\) and \(\text{nbLoans[mId]}\) can be optimized based on the simple observation that attribute variables \(\text{borrower[bId']}\) for \(\text{bId'} \neq bId\) and \(\text{nbLoans[mId']}\) for \(\text{mId'} \neq mId\) remain unaffected. Hence, the corresponding part of process \(M\) regarding \(\text{Eb3}^3\) action “\(\text{Lend (bId, mId)}\)” can be modified as follows:

\[
\begin{align*}
\text{Lend (bId, mId, nbLoans, borrower);} \\
\text{ borrower [ord (bId)] := mId}; \\
\text{ nbLoans [ord (mId)] := nbLoans [ord (mId)] + 1}
\end{align*}
\]

A similar approach applies to communication label \(\text{Ret}\). The optimized LNT specification of the library management system is given in the appendix. Note that this optimization technique is applicable whenever the \text{in} offers of the communication label in question, i.e. the parameters marked with ?, suffice to determine
and every process definition of the form "P" is given in Figure 9. An evaluation is delayed until the first action occurring in the process expression. This led us to consider a second parameter for the translation function \( t \).

We define a translation function \( t \) from \( \mathcal{EB}^3 \) to LNT process algebra constructs with a direct correspondence in LNT. The main difficulty arises in the translation of guarded process expressions of the form "\( \alpha (\overline{\pi}, \lambda) \)". The Rules of Figure 9 can be commented as follows:

**Rule (1)** translates the \( \lambda \) action. Note that \( \lambda \) cannot be translated to the empty LNT statement \( \text{null} \), because execution of \( \lambda \) 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 $\lambda (\text{?f})$. The guard $C$ is evaluated after replacing calls to attribute functions (all of which have the form $f_i(T, v_i)$) by the appropriate attribute variables, using function $\text{mod}$ defined in Figure 7. Rule (2) is similar but handles visible actions.

- Rule (3) translates $\text{EB}^3$ 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 $\text{EB}^3$ into their direct LNT counterpart.

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

- Rule (8) translates $\text{EB}^3$ parallel composition into LNT parallel composition.

- Rule (9) translates $\text{EB}^3$ 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 $\text{EB}^3$ 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, 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 $\text{pr}_C$ (defined in Figure 10) is placed in parallel to $t(E_0, \text{true})$ and both processes synchronize on all actions. Process $\text{pr}_C$ imposes on $t(E_0, \text{true})$ the constraint that the first executed action must satisfy the condition $C$ (then branch). For subsequent actions, the condition is relaxed (else branch).

\[\text{head normal form}\] [BPS01], is used principally in the context of the process algebra ACP [BK85] to analyse the behaviour of recursive processes.
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] (b1)
    end select
  end loop
  par
    loop L in
      select break L [] member [Register, Unregister, Lend, Return] (m1)
    end select
  end loop
  par
    loop L in
      select break L [] member [Register, Unregister, Lend, Return] (m2)
    end select
  end loop
end par
```
Verification of EB³ Specifications using CADP

Note that the equivalent LNT process of EB³ process \texttt{main} is placed in parallel to memory process \texttt{M}. Moreover, the simulation of EB³ quantified parallel synchronization operator (that has no equivalent operator in LNT) is applied to EB³ expressions “\[ | | bId : BID : book (bId) * \]” and “\[ | | mId : MID : member (mId) * \]” and the EB³ Kleene Closure operator is simulated as described in Figure 9.

The body of LNT processes \texttt{book} and \texttt{member} is defined as below:

\begin{verbatim}
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
\end{verbatim}

The definition of LNT process \texttt{member} depends on LNT process \texttt{loan}, whose definition is given below:

\begin{verbatim}
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, ?borrower)
        where ((borrower [ord (bId)] eq m⊥) and (nbLoans [ord (mId)] eq 1));
    Return (bId)
    end var
end process
\end{verbatim}

The following example illustrates and justifies the use of process \texttt{prC} as a means to solve the guard-action atomicity problem. Consider the EB³ system:

\[ C \Rightarrow \text{Lend} (b_1, m_1) \ || \ Return (b_2), \]

where \( C \) denotes the Boolean condition:

\[ \text{borrower} (T, b_1) = \bot \land \text{nbLoans} (T, m_1) < \text{NbLoans} \]

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

\begin{verbatim}
par \text{Lend}, \text{Return}, \lambda \ in
    par \text{Lend}, \text{Return}, \lambda \ in
        par \text{Lend} (b_1, m_1, ?j) \ || \ Return (b_2, ?j) end par
    || \text{prC} [\text{Lend}, \text{Return}, \lambda] (b_1, m_1)
end par

|| \text{M} [\text{Lend}, \text{Return}, \lambda]
end par
\end{verbatim}

The first action executed by this system may be either \text{Lend} or \text{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 (b₁, m₁, ?f)” in the above expression,
- “Lend (b, m, ?f) where borrower[ord(b₁)] = ⊥ ∧ nbLoans[ord(m₁)] < NbLoans” in process prC (at this moment, start is true, see Fig. 10), and
- “Lend (b, m, !f)” in process M (see Fig. 7).

Thus, in prC at synchronization time, f is an up-to-date copy of the memory stored in M, b = b₁, and m = m₁. The only condition for the synchronization to occur is the guard mod (C), whose value is computed using the up-to-date copy f of the memory. In case mod (C) evaluates to true, no other action (susceptible to modifying 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 EB³ specifications to LNT, named EB³2LNT, implemented using the Ocaml Lex/Yacc compiler construction technology. It consists of about 900 lines of OCaml code. We applied EB³2LNT on a benchmark of EB³ specifications, which includes variations of the library management system.

We noticed that, for each EB³ 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 EB³: 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 E₀ of the quantified parallel composition operator “[[Δ] x:V : E₀]” as many times as there are elements in the set V. Both expansions are linear in the size of the source EB³ code. However, in the case of a nested parallel composition “[[Δ₁] x₁:V₁ : . . . [Δₙ] xₙ:Vₙ : E₀]”, the expansion factor is as high as the product of the number of elements in the respective sets V₁, . . . , Vₙ, which may be large. If E₀ is a big process expression, the expansion can be limited by encapsulating E₀ in a parameterized process “P_E₀(x₁, . . . , xₙ)” and replacing duplicated occurrences of E₀ by appropriate instances of P_E₀.

**10. Case Study Revisited**

We illustrate below the application of the EB³2LNT translator in conjunction with CADP for analyzing an extended version of the IS library management system, whose description in EB³ 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.
Verification of EB³ Specifications using CADP

Table 1. Model checking results for the library management system

\[ \begin{array}{|c|c|c|c|c|} \hline
(m,p) & (3,2) & (3,3) & (3,4) & (4,3) \\
\hline
\text{states} & 1,002 & 182,266 & 8,269,754 & 27,204,016 \\
\hline
\text{trans.} & 5,732 & 1,782,348 & 105,481,364 & 330,988,232 \\
\hline
\text{time} & 1.9s & 14.4s & 31'39s & 140'22s \\
\hline
\text{R1} & 0.3s & 1.8s & 5'19s & 20'13s \\
\hline
\text{R2} & 0.2s & 2.9s & 9'26s & 36'7s \\
\hline
\text{R3} & 0.2s & 9.4s & 26'47s & 26'7s \\
\hline
\text{R4} & 0.2s & 5'15s & 18'49s & 36'07s \\
\hline
\text{R5} & 0.2s & 2.2s & 6'46s & 21'52s \\
\hline
\text{R6} & 0.2s & 4.1s & 38'30s & 10'19s \\
\hline
\text{R7} & 0.2s & 7.4s & 65'22s & 24'33s \\
\hline
\text{R8} & 0.2s & 2.3s & 6'38s & 22'29s \\
\hline
\text{R9} & 0.2s & 2.3s & 6'38s & 22'29s \\
\hline
\text{R10} & 0.3s & 13.3s & 43'59s & 62'07s \\
\hline
\text{R11} & 0.3s & 2.5s & 6'36s & 22'14s \\
\hline
\text{R12} & 0.3s & 4.0s & 10'47s & 45'09s \\
\hline
\text{R13} & 0.4s & 4.3s & 11'46s & 36'07s \\
\hline
\text{R14} & 0.3s & 3.6s & 10'41s & 37'33s \\
\hline
\text{R15} & 0.2s & 2.8s & 7'53s & 28'56s \\
\hline
\end{array} \]

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:

\[
[\text{true}^*; \{\text{ACQUIRE} ?B : \text{string}\}.(\not \{\text{DISCARD} !B\})^*; \{\text{ACQUIRE} !B\}] \text{ false}
\]

This formula uses the standard safety pattern “\([\beta] \text{ false}\)”, which forbids the existence of transition sequences matching the regular formula \(\beta\). 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 \(\text{true}^*\) matches a subsequence of (zero or more) transitions labeled by arbitrary actions. Note the use of the construct “?\(B : \text{string}\)”, 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:

\[
[\text{true}^*; \{\text{RESERVE} ?M : \text{string} ; ?B : \text{string}\}.(\not (\{\text{TAKE} !M !B\} \text{ or } \{\text{TRANSFER} !M !B\})^*]
\]

\[
(\not (\{\text{TAKE} !M !B\} \text{ or } \{\text{TRANSFER} !M !B\})^*). \{\text{CANCEL} !M !B\}] \text{ true}
\]

This formula denotes a liveness property of the form “[\(\beta_1\) \langle \(\beta_2\) true\]”, which states that every transition sequence matching the regular formula \(\beta_1\) (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 \(\beta_2\) (in this case, the reservation can be cancelled before being taken or transferred).

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

Using EB³2LNT, we translated the EB³ 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 $\text{NbLoans} = 1$, 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 $\text{eb32lnt}$, 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 $\text{eb3}$ method with formal verification capabilities by reusing already available model checking technology. Our approach relies upon a new translation from $\text{eb3}$ 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 $\text{eb3}$ [VD13] instead of the original trace semantics [FS03], was automated by the $\text{eb32lnt}$ translator and validated on several examples of typical ISs. So far, we experimented only the model checking of MCL data-based temporal properties on $\text{eb3}$ 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 $\text{eb3}$ to LNT that can be found in [Vek14], which could serve as reference for translating $\text{eb3}$ to other process algebras as well.

$\text{eb32lnt}$ 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 $\text{eb3}$ 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 $\text{eb3}$ 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 $\text{eb3}$ 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 $\text{eb3}$ specifications.
References


---

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
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 $mId : mld$, $bId : BID$, acquired : $ACQUIR$, borrower : $BOR$, nbLoans : $NB$, reservation : $RESERV$ in
acquired := $ACQUIR$(false); borrower := $BOR$(m_\perp);
bLoans := $NB$(0); reservation := $RESERV$(NIL);

loop
select
Acquire (?bId);
acquired [ord (bId)] := true
[] Discard (?bId, !borrower, !reservation);
acquired [ord ()] := false
[] Register (?mId);
bLoans [ord (mId)] := 1
[] Unregister (?mId !nbLoans !reservation)
bLoans [ord (mId)] := 0
[] Lend (?bId, ?mId, !acquired, !borrower, !reservation, !nbloans);
borrower [ord (bId)] := mId;
bLoans [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;
bLoans [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);
bLoans [ord (borrower [ord (bId)])] := nbLoans [ord (borrower [ord (bId)])] - 1;
borrower [ord (bId)] := mId;
bLoans [ord (mId)] := nbLoans [ord (mId)] + 1
[] Renew (?bId, !acquired, !borrower, !reservation)
[] Return (?bId, !acquired, !borrower);
bLoans [ord (borrower [ord (bId)])] := nbLoans [ord (borrower [ord (bId)])] - 1;
borrower [ord (bId)] := m_\perp

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 $m_\perp$) 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 mId) 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
            (nbLoans[ord (mId)] eq 0) and (nbLoans[ord (mId')] gt 2) 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 mId) 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')] eq 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)] eq 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 mId) and
            (nbLoans[ord (mId')] eq 0) 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 mId) 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')] eq 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 $L_1$ in select break $L_1$ []
| book [Acquire, Discard] ($b_2$)
| end select end loop
| end par
||
| par
| loop $L_1$ in select break $L_1$ []
| Register($m_1$);
| par
| loop $L_2$ in select break $L_2$ []
| member [Lend, Take, Renew, Return, Reserve, Cancel, Transfer] ($m_1, b_1$)
| end select end loop
||
| loop $L_2$ in select break $L_2$ []
| member [Lend, Take, Renew, Return, Reserve, Cancel, Transfer] ($m_1, b_2$)
| end select end loop;
| end par;
| var nbLoans : NB, bid : BID, reservation : RESERV in
| Unregister ($m_1$, ?nbLoans, ?reservation)
| where ((nbLoans $|$ ord ($m_1$)) eq 1) and
| (is_reserved ($m_1$, reservation $|$ ord ($b_1$))) eq false) and
| (is_reserved ($m_1$, reservation $|$ ord ($b_2$))) eq false)
| end var
| end select end loop
||
| loop $L_1$ in select break $L_1$ []
| Register($m_2$);
| par
| loop $L_2$ in select break $L_2$ []
| member [Lend, Take, Renew, Return, Reserve, Cancel, Transfer] ($m_2, b_1$)
| end select end loop
||
| loop $L_2$ in select break $L_2$ []
| member [Lend, Take, Renew, Return, Reserve, Cancel, Transfer] ($m_2, b_2$)
| end select end loop
| end par;
| var nbLoans : NB, bid : BID, reservation : RESERV in
| Unregister ($m_2$, ?nbLoans, ?reservation)
| where ((nbLoans $|$ ord ($m_2$)) eq 1) and
| (is_reserved ($m_2$, reservation $|$ ord ($b_1$))) eq false) and
| (is_reserved ($m_2$, reservation $|$ ord ($b_2$))) 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”

\[
\text{macro } R_1(B) = \begin{align*}
\left( \left[ \left( \text{not} \left\{ \text{ACQUIRE} \not B \right\} \right)^* \left\{ \text{ACQUIRE} \not B \right\} \text{ true} \right. \right. \\
\text{ and } \\
\left. \left. \text{true}^* \left\{ \text{DISCARD} \not B \right\} \left( \text{not} \left\{ \text{ACQUIRE} \not B \right\} \right)^* \left\{ \text{ACQUIRE} \not B \right\} \text{ true} \right) \end{align*}
\]
end_macro
\]

\[ R_1("B1") \text{ and } R_1("B2") \text{ and } R_1("B3") \]

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”

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

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

\[
\text{macro } R_4(M) = \begin{align*}
\left( \left[ \left( \text{not} \left\{ \text{JOIN} ! M \right\} \right)^* \left( \left\{ \text{LEND} ! M \ ?any : \text{string} \right\} \text{ or } \left\{ \text{TAKE} ! M \ ?any : \text{string} \right\} \right) \text{ false} \right. \right. \\
\text{ and } \\
\left. \left. \text{true}^*. \left\{ \text{LEAVE} ! M \right\} . \left( \text{not} \left\{ \text{JOIN} ! M \right\} \right)^* . \\
\left( \left\{ \text{LEND} ! M \ ?any : \text{string} \right\} \text{ or } \left\{ \text{TAKE} ! M \ ?any : \text{string} \right\} \right) \text{ false} \right) \end{align*}
\]
end_macro
\]

\[ R_4("M1") \text{ and } R_4("M2") \text{ and } R_4("M3") \]

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 ?$M_2$ : 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”
Verification of EB^3 Specifications using CADP

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

\[
\text{true}. \{\text{RESERVE } M^! B \}. \{\text{not } \{\text{TAKE } M^! B \} \text{ or } \{\text{CANCEL } M^! B \}\}. \{\text{RENEW } B \}. \text{false}
\]

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"

\[
\text{true}. \{\text{LEND } M^! B \text{ or } \text{TAKE } M^! B \}. \{\text{not } \{\text{RETURN } B \}\}. \{\text{not } \{\text{LEND } M^! B \} \text{ or } \{\text{TAKE } M^! B \}\}. \{\text{not } \{\text{RETURN } B \}\}. \{\text{RETURN } B \}. \text{false}
\]

This property corresponds to the pattern "\( \alpha_1 \) does not occur between \( \alpha_2 \) and \( \alpha_3 \)\), which is expressed by the following scheme, easily recognizable in this formula:

\[
\text{true}. \alpha_2. \{\text{not } \alpha_3\}. \alpha_1. \{\text{not } \alpha_3\}. \alpha_3 \text{ false}
\]

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

\textbf{Requirement R14} “Ultimately, there is always a procedure that enables a member to leave the library”

\textbf{macro} \ R_{14} (M) = \\
\quad \{ \\
\quad \quad \text{true}. \{ \text{JOIN } ! M \}. (\text{not } (\{ \text{LEAVE } ! M \}). ((\text{not } (\{ \text{LEAVE } ! M \}). \{ \text{LEAVE } ! M \}) \text{ true} \\
\quad \} \text{ end_macro} \\
R_{14} ("M1")

\textbf{Requirement R15} “A member cannot borrow more than the loan limit defined at the system level for all users”

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

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