Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus
Radu Mateescu, Mihaela Sighireanu

To cite this version:

HAL Id: inria-00072755
https://inria.hal.science/inria-00072755
Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus

Radu Mateescu — Mihaela Sighireanu

N° 3899
Mars 2000

THÈME 1
Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus

Radu Mateescu*, Mihaela Sighireanu**

Thème 1 — Réseaux et systèmes
Projet VASY

Rapport de recherche n° 3899 — Mars 2000 — 24 pages

Abstract: Model-checking is a successful technique for automatically verifying concurrent finite-state systems. When building a model-checker, a good compromise must be made between the expressive power of the property description formalism, the complexity of the model-checking problem, and the user-friendliness of the interface. We present a temporal logic and an associated model-checking method that attempt to fulfill these criteria. The logic is an extension of the alternation-free $\mu$-calculus with ACTL-like action formulas and PDL-like regular expressions, allowing a concise and intuitive description of safety, liveness, and fairness properties over labeled transition systems. The model-checking method is based upon a succinct translation of the verification problem into a boolean equation system, which is solved by means of an efficient local algorithm having a good average complexity. The algorithm also allows to generate full diagnostic information (examples and counterexamples) for temporal formulas. This method is at the heart of the EVALUATOR 3.0 model-checker that we implemented within the CADP toolset using the generic OPEN/CAESAR environment for on-the-fly verification.

Key-words: boolean equation system, diagnostic, labelled transition system, model-checking, $\mu$-calculus, specification, temporal logic, verification


* Radu.Mateescu@inria.fr
** Mihaela.Sighireanu@liafa.jussieu.fr
Evaluation efficace à la volée pour le mu-calcul régulier sans alternance

Résumé : La vérification basée sur les modèles (model-checking) est une technique utilisée avec succès pour la vérification automatique des systèmes concurrents à états finis. Lors de la construction d’un évaluateur (model-checker), il est nécessaire d’effectuer un bon compromis entre l’expressivité du formalisme de description des propriétés, la complexité du problème de la vérification et la facilité d’utilisation de l’interface. Nous présentons une logique temporelle et une méthode de vérification associée conçues afin de satisfaire ces critères. La logique est une extension du μ-calcul sans alternance avec des formules sur actions comme en ACTL et des expressions régulières comme en PDL, permettant une description concise et intuitive des propriétés de sûreté, vivacité et équité sur des systèmes de transitions étiquetées. La méthode de vérification est basée sur une traduction succincte du problème vers un système d’équations booléennes qui est résolu au moyen d’un algorithme efficace ayant une bonne complexité moyenne. L’algorithme permet aussi de générer des diagnostics (exemples et contre-exemples) pour les formules temporelles. Cette méthode sert de base à l’évaluateur EVALUATOR 3.0 que nous avons implémenté dans la boîte à outils CADP en utilisant l’environnement générique de vérification à la volée OPEN/CAESAR.

Mots-clés : diagnostic, logique temporelle, mu-calcul, spécification, système d’équations booléennes, système de transitions étiquetées, vérification basée sur les modèles
1 Introduction

Formal verification is essential in order to improve the reliability of complex, critical applications such as communication protocols and distributed systems. A state-of-the-art technique for automatic verification of concurrent finite-state systems is called model-checking. In this approach, the application under design is first translated into a finite labeled transition system (LTS) model, on which the desired correctness properties (expressed e.g., as temporal logic formulas) are verified using appropriate model-checking algorithms.

When designing and building a model-checker, several important criteria must be considered. Firstly, the specification formalism should be sufficiently powerful to describe the main temporal property classes usually encountered (safety, liveness, fairness). Among the wide range of temporal logics proposed in the literature, the modal $\mu$-calculus [18] is particularly powerful, subsuming linear-time logics as LTL [22], branching-time logics as CTL [4] or ACTL [25], and regular logics as PDL [12] or PDL-$\Delta$ [27].

Secondly, the underlying model-checking problem should have a sufficiently low complexity, in order to offer reasonable response times on practical applications. Optimizing this is often contradictory with the first criterion above, because the model-checking complexity of temporal logics usually increases with their expressive power. Since the model-checking problem of the full $\mu$-calculus is exponential-time, various sublogics of lower complexity have been defined. Among these, the alternation-free fragment [7] makes a good compromise between expressiveness (allowing direct encodings of CTL and ACTL) and efficiency of model-checking (several linear-time algorithms being available [5,1,29,20]).

Thirdly, the model-checker interface should allow an intuitive, concise, and flexible description of properties, in order to avoid specification errors and to facilitate the verification task for non-expert users. Moreover, the model-checker must provide enough feedback information to make the debugging of the applications feasible; in practice, this means to provide a precise diagnostic in addition to a simple yes/no answer for a temporal property.

In this paper, we present a temporal logic and an associated model-checking method attempting to fulfill the aforementioned criteria. The temporal logic adopted is an extension of the alternation-free $\mu$-calculus with ACTL-like action formulas and PDL-like regular expressions, allowing a concise and intuitive description of safety, liveness, and (some) fairness properties without sacrificing the efficiency of verification. The method proposed for verifying a temporal formula over an LTS has a linear-time worst-case complexity (both in LTS size and formula size) and is based upon a succinct translation of the verification problem into a boolean equation system (BES). The method works on-the-fly, by exploring the LTS in a demand-driven way during the verification of the formula. The resulting BES is solved using a new linear-time local algorithm based on a depth-first search of the corresponding boolean graph. Compared to other linear-time local algorithms [1,29], our algorithm is simpler to understand and has a good average complexity, achieved by a careful bookkeeping of the information in the portion of boolean graph visited during the search. Moreover, our algorithm is easily connected to the diagnostic generation algorithms given in [24], allowing to produce examples and counterexamples (subgraphs of the LTS) fully explaining the truth values of the formulas. This verification method has been used as a basis for the EVALUATOR 3.0
model-checker that we developed within the CADP (ÇÆŠAR/AŁDÉBARAN) toolset [9] using
the generic OPEN/ÇÆŠAR environment for on-the-fly verification [13].

The paper is organized as follows. Section 2 defines the syntax and semantics of the
temporal logic proposed and illustrates its use by means of various examples of properties.
Section 3 presents in detail the model-checking method and Section 4 discusses its imple-
mentation within the CADP toolset. Finally, Section 5 gives some concluding remarks and
directions for future work.

2 Regular alternation-free µ-calculus

The logic that we propose, called regular alternation-free µ-calculus, is an extension of the
alternation-free fragment of the modal µ-calculus [18,7] with action formulas as in ACTL [25]
and with regular expressions over action sequences as in PDL [12]. It allows direct encodings
of “pure” branching-time logics like ACTL or CTL [4], as well as of regular logics like PDL
or PDL-∆ [27]. We first define its syntax and semantics, and then we show its usefulness by
means of several examples of commonly encountered temporal properties.

2.1 Syntax and semantics

We consider as interpretation models finite labeled transition systems (LTSs), which are
particularly suitable for action-based description formalisms such as process algebras. An
LTS is a tuple L = (S, A, T, s0), where: S is a finite set of states, A is a finite set of actions, T ⊆
S × A × S is the transition relation, and s0 ∈ S is the initial state. A transition (s, a, s′) ∈ T,
also noted s → a s′, indicates that the system can move from state s to state s′ by performing
action a.

The regular alternation-free µ-calculus is built from three types of formulas, according to
the syntax given on Figure 1.

| Action formulas | α ::= a | ¬α | α1 ∧ α2 |
| Regular formulas | β ::= α | β₁ β₂ | β₁ β₂ | β* |
| State formulas | ϕ ::= F | T | ϕ₁ ∨ ϕ₂ | ϕ₁ ∧ ϕ₂ | ⟨β⟩ ϕ | [β] ϕ | Y ϕ | µY.ϕ | νY.ϕ |

Fig. 1. Syntax of regular alternation-free µ-calculus

Action formulas α are built from action names a ∈ A by using the standard boolean
operators. Derived boolean connectives are defined as usual: F = a ∧ ¬a for some a, T = ¬F,
α₁ ∨ α₂ = ¬(¬α₁ ∧ ¬α₂), etc. Regular formulas β are built from action formulas α by using the
standard regular expression operators: concatenation (.), choice (l), and transitive-reflexive
Efficient Model-Checking for Regular Mu-Calculus

closure (*). The empty sequence operator ε and the transitive closure operator + are defined as ε = F* and β* = ββ*. State formulas φ are built from propositional variables $Y \subseteq Y$ by using the standard boolean operators, the possibility and necessity operators $\langle \beta \rangle \phi$ and $[\beta] \phi$, and the minimal and maximal fixed point operators $\mu Y.\phi$ and $\nu Y.\phi$. The $\mu$ and $\nu$ operators act as binders for $Y$ variables in a way similar to quantifiers in first-order logic. A formula $\phi$ without free occurrences of $Y$ variables is closed. Formulas $\phi$ are assumed to be alternation-free, i.e., without mutually recursive minimal and maximal fixed point subformulas ($\langle \beta \rangle \phi'$ and $[\beta] \phi'$ modalities, where $\beta$ contains $*$ operators, must be considered as "hidden" minimal and maximal fixed point subformulas, respectively).

![Table: Semantics of regular alternation-free $\mu$-calculus](image)

**Fig. 2.** Semantics of regular alternation-free $\mu$-calculus

The semantics of the logic is shown on Figure 2. The interpretation $[\alpha] \subseteq A$ of action formulas gives the set of LTS actions satisfying $\alpha$. The interpretation $[\beta] \subseteq S \times S$ of regular formulas gives a binary relation between the source and target states of transition sequences satisfying $\beta$ (\&, and $*$ denote composition, union, and transitive-reflexive closure of binary relations). The $\alpha$ regular formula characterizes one-step sequences $s \xrightarrow{\alpha} s'$ such that $a$ satisfies $\alpha$. The $\beta_1$, $\beta_2$ formula states that a sequence is the concatenation of two sequences satisfying $\beta_1$ and $\beta_2$; $\beta_1 | \beta_2$ states that a sequence can satisfy $\beta_1$ or $\beta_2$; and $\beta^*$ states that a sequence is the concatenation of (zero or more) sequences satisfying $\beta$. The interpretation $[\phi] \rho \subseteq S$ of state formulas, where the propositional context $\rho : Y \rightarrow 2^S$ assigns state sets to propositional...
variables, gives the set of LTS states satisfying \( \varphi \) in the context of \( \rho \) (\( \otimes \) denotes context overriding). The modalities \( \langle \beta \rangle \varphi \) and \( \lbrack \beta \rbrack \varphi \) characterize the states for which some (all) outgoing transition sequences satisfying \( \beta \) lead to states satisfying \( \varphi \). The formulas \( \mu Y . \varphi \) and \( \nu Y . \varphi \) denote the minimal and maximal solutions (over \( 2^S \)) of the fixed point equation \( Y = \varphi \).

Let \( L = (S, A, T, s_0) \) be an LTS. An action \( a \in A \) satisfies a formula \( \alpha \) (written as \( a \models \alpha \)) iff \( a \in \llbracket \alpha \rrbracket \). A state \( s \in S \) satisfies a closed formula \( \varphi \) (written \( s \models \varphi \)) iff \( s \in \llbracket \varphi \rrbracket \). \( L \) is a \( \varphi \)-model (written \( L \models \varphi \)) iff \( \llbracket \varphi \rrbracket = S \). Since an on-the-fly model-checker only decides whether \( s_0 \models \varphi \), the user should be aware that verifying \( L \models \varphi \) amounts to check on-the-fly the formula \( [T^*] \varphi \) (equivalent to the ACTL formula \( A[\neg \text{Grant}] \varphi \)), stating that \( \varphi \) holds on every state reachable from \( s_0 \).

### 2.2 Examples

The regular alternation-free \( \mu \)-calculus allows to express intuitively and concisely various useful properties of LTSs. Table 1 shows several examples of typical formulas representing safety, liveness, and fairness properties.

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PROPERTY</th>
<th>FORMULA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety</td>
<td>Absence of Error actions</td>
<td>([T^* \text{Error}] F)</td>
</tr>
<tr>
<td></td>
<td>Unreachability of a ( \text{Recv} ) action before a ( \text{Send} )</td>
<td>([\neg \text{Send}]^* . \text{Recv}] F)</td>
</tr>
<tr>
<td></td>
<td>Mutual exclusion of sections delimited by ( \text{Open} ) and ( \text{Close} )</td>
<td>([T^* . \text{Open} . (\neg \text{Close})]^* . \text{Open}] F)</td>
</tr>
<tr>
<td>Liveness</td>
<td>Deadlock freedom: absence of states without successors</td>
<td>([T^*] (T) T)</td>
</tr>
<tr>
<td></td>
<td>Potential reachability (via some ( \text{Errors} )) of a ( \text{Recv} ) after a ( \text{Send} )</td>
<td>([T^* . \text{Send} (T . \text{Error})]^* . \text{Recv}] T)</td>
</tr>
<tr>
<td></td>
<td>Inevitable reachability of a ( \text{Grant} ) action after a ( \text{Request} )</td>
<td>([T^* . \text{Request}] \mu Y . (T) \wedge (\neg \text{Grant}) Y)</td>
</tr>
<tr>
<td>Fairness</td>
<td>Livelock freedom: absence of ( \text{tau} )-circuits</td>
<td>([T^*] \mu Y . [\text{tau}] Y)</td>
</tr>
<tr>
<td></td>
<td>Fair reachability (by skipping circuits) of a ( \text{Recv} ) after a ( \text{Send} )</td>
<td>([T^* . \text{Send} (\neg \text{Recv})]^* \langle (\neg \text{Recv})^* . \text{Recv} \rangle T)</td>
</tr>
</tbody>
</table>

Note that boolean connectives (negation in particular) over actions improve the conciseness of formulas: without these operators, it would be impossible to express the inevitable
reachability of an action without referring to other actions in the LTS. Also, regular operators (although theoretically they do not increase the expressive power of the alternation-free modal $\mu$-calculus) improve the readability of formulas: without these operators, the second liveness property given in Table 1 would be described by the equivalent fixed point formula $\mu Y_1.(\langle \text{Send} \rangle \mu Y_2.(\langle \text{Recv} \rangle \top \lor \mu Y_3.(\langle \text{Error} \rangle Y_2 \lor (\top) Y_3)) \lor (\top) Y_1).

Other, more elaborate examples of generic temporal properties encoded in regular alternation-free $\mu$-calculus can be found in Section 4.

3 On-the-fly model-checking

We present in this section a method for on-the-fly model-checking of regular alternation-free $\mu$-calculus formulas over finite LTSs. The method works by translating the verification problem into a boolean equation system, which is simultaneously solved using an efficient local algorithm.

3.1 Translation into boolean equation systems

Consider an LTS $L = (S, A, T, s_0)$ and a closed formula $\varphi$ in normal form (i.e., in which all propositional variables are unique). The verification problem we are interested in consists of deciding whether $s_0 \models \varphi$. An efficient method used for the ACTL logic [8] and for the alternation-free $\mu$-calculus $[5,1]$ is to translate the problem into a boolean equation system (BES) $[1,21]$, which is solved using specific local algorithms $[1,29,28]$. For the regular alternation-free $\mu$-calculus, one way to proceed could be first to translate a state formula $\varphi$ in plain alternation-free $\mu$-calculus and then to apply the above procedure. This means to encode the regular modalities of $\varphi$ using fixed point operators, e.g., by applying the Emerson-Lei translation from PDL to alternation-free $\mu$-calculus $[7]$. This translation is succinct (it produces at most a linear blow-up in the size of $\varphi$), but requires the identification and sharing of common subformulas.

However, we can also devise a succinct translation of the verification problem $s_0 \models \varphi$ into a BES resolution without computing common subformulas, but using instead an equational intermediate representation. The translation that we propose involves three steps, described below.

Translation into PDL with recursion. The first step is to translate a regular alternation-free $\mu$-calculus formula $\varphi$ into PDL with recursion (PDLR), which is a generalization of the Hennessy-Milner logic with recursion $\text{HMLR}$ $[19]$. A PDLR specification (see Figure 3) consists of a propositional variable $Y$ and a fixed point equation system with propositional variables in left-hand sides and PDL formulas in right-hand sides. The equation system is given as a list $M_1, \ldots, M_p$ of $\sigma$-blocks $(\cdot$ denotes concatenation), i.e., subsystems of equations with the same sign $\sigma \in \{\mu, \nu\}$. We consider here only alternation-free PDLR specifications, in which every $\sigma$-block $M_j$ (for $1 \leq j < p$) depends only upon (has free variables that may be
bound in) \( M_{j+1}, \ldots, M_p \). The \( Y \) variable must be defined in one of the \( \sigma \)-blocks \( M_1, \ldots, M_p \) (usually in \( M_1 \)). A PdlR specification is \emph{closed} if all variables occurring in it are bound in the equation system.

**Syntax of a PdlR specification:**
\[
P = (Y; M_1, \ldots, M_p)
\]
where \( M_j = \{ Y_{j_i}, \varphi_j \}_{1 \leq i \leq n_j} \) for all \( 1 \leq j \leq p \)

**Semantics w.r.t. an Lts \((S, A, T, s_0)\) and a context \( \rho : Y \to 2^S \):**
\[
\begin{align*}
([Y; M_1, \ldots, M_p]) \rho &= (\rho \circ [M_1, \ldots, M_p]) \rho(Y) \\
[M_j] \rho &= ([M_j] (\rho \circ [M_{j+1}, \ldots, M_p]) \rho), [M_{j+1}, \ldots, M_p] \rho \\
[[Y_{j_i}, \varphi_j]] \varphi \rho(Y_{j_1}, \ldots, Y_{j_{n_j}}) &= [\sigma_j \bar{\mathcal{F}}_j(\rho(U_1, \ldots, U_{n_j}))](\varphi_j(Y_{j_1}, \ldots, Y_{j_{n_j}}))_{1 \leq i \leq n_j}
\end{align*}
\]
where \( \bar{\mathcal{F}}_j : (2^S)^n \to (2^S)^n \), \( \bar{\mathcal{F}}(U_1, \ldots, U_{n_j}) = (\varphi_j(U_i, \ldots, U_{n_j}})_{1 \leq i \leq n_j} \)

**Fig. 3. Syntax and semantics of PdlR**

A PdlR specification \((Y; M_1, \ldots, M_p)\) interpreted over an Lts yields the set of states associated to \( Y \) in the solution of \( M_1, \ldots, M_p \). The solution of \( M_1, \ldots, M_p \) is a propositional context in \( Y \to 2^S \) obtained by concatenating the solutions of all \( \sigma \)-blocks \( M_j \) (\( 1 \leq j < p \)), each one being calculated in the context of the subsystem \( M_{j+1}, \ldots, M_p \). The solution of a \( \sigma \)-block \( M_j \) with \( n_j \) variables is a context mapping \( M_j \)'s variables to the \( \sigma_j \) fixed point of an associated vectorial functional defined over \((2^S)^{n_j}\). The semantics of an empty system \( \{ \} \) is the empty context \([ \].

Before translating a closed regular alternation-free \( \mu \)-calculus formula \( \varphi \) in PdlR, we must convert \( \varphi \) into \emph{expanded} form, by performing two actions: (a) add a new \( \mu Y \) (\( \nu Y \)) operator, where \( Y \) is a “fresh” variable, in front of every \( (\beta) \mathcal{V} \) (\( \beta \mathcal{V} \)) subformula of \( \varphi \) in which \( \beta \) contains a \( * \) operator (recall from Section 2.1 that these modalities are considered as “hidden” fixed point operators); (b) if the resulting formula \( \varphi_0 \) is not a fixed point one, add in front of \( \varphi_0 \) a \( \sigma Y_0 \) operator, where \( \sigma \in \{ \mu, \nu \} \) and \( Y_0 \) is another “fresh” variable.

The translation of an expanded formula \( \sigma Y_0 \varphi_0 \) into a PdlR specification \((T_1(\sigma Y_0 \varphi_0, \sigma), T_2(\sigma Y_0 \varphi_0, \sigma))\) is obtained using two syntactic functions \( T_1 \) and \( T_2 \), defined inductively in Figure 4. \( T_1(\varphi, \sigma) \) yields a formula obtained from \( \varphi \) by substituting each fixed point subformula by its corresponding variable, \( T_2(\varphi, \sigma) \) yields a system containing, for each fixed point subformula of \( \varphi \), an equation with the corresponding variable in the left-hand side and a PdlR formula in the right-hand side. The first \( \sigma \)-block, denoted by \( h(\mathcal{T}_2(\varphi, \sigma)) \), contains the equations of sign \( \sigma \) associated to the topmost fixed point subformulas of \( \varphi \). The remainder of the system, denoted by \( t(\mathcal{T}_2(\varphi, \sigma)) \), contains the \( \sigma \)-blocks already constructed from subformulas of \( \varphi \). A new \( \sigma \)-block is created every time that a fixed point subformula with a sign \( \sigma \) dual to \( \sigma \) is encountered (\( \mu = \nu \) and \( \nu = \mu \)).
<table>
<thead>
<tr>
<th>( \varphi )</th>
<th>( T_1(\varphi, \sigma) )</th>
<th>( T_2(\varphi, \sigma) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
<tr>
<td>( \neg \varphi_1 )</td>
<td>( \neg \varphi_1 )</td>
<td>( \neg \varphi_1 )</td>
</tr>
<tr>
<td>( \varphi_1 \land \varphi_2 )</td>
<td>( T_1(\varphi_1, \sigma) \land T_1(\varphi_2, \sigma) )</td>
<td>( T_2(\varphi_1, \sigma) \land T_2(\varphi_2, \sigma) )</td>
</tr>
<tr>
<td>( \varphi_1 \lor \varphi_2 )</td>
<td>( T_1(\varphi_1, \sigma) \lor T_1(\varphi_2, \sigma) )</td>
<td>( (\text{hd}(T_2(\varphi_1, \sigma)) \cup \text{hd}(T_2(\varphi_2, \sigma))) )</td>
</tr>
<tr>
<td>( \sigma Y. \varphi_1 )</td>
<td>( Y )</td>
<td>( { } )</td>
</tr>
<tr>
<td>( \bar{\sigma} Y. \varphi_1 )</td>
<td>( { } )</td>
<td>( { } )</td>
</tr>
</tbody>
</table>

**Fig. 4.** Translation of state formulas in PDLR

We illustrate this translation by an example. Consider the following formula (already written in expanded form), stating that every \( \text{Send} \) action in the \( \text{LTS} \) will be eventually followed by a \( \text{Recv} \):

\[
\varphi = \nu Y_0.\left[T^* . \text{Send}\right] Y_1 . (\top) \land \neg \text{Recv} \ Y_1
\]

The translation \( (T_1(\varphi, \nu), T_2(\varphi, \nu)) \) yields the PDLR specification below:

\[
\{ Y_0, Y_0 \overset{\nu}{\leftarrow} \left[T^* . \text{Send}\right] Y_1 \}, \{ Y_1 \overset{\top}{\leftarrow} \land \neg \text{Recv} \ Y_1 \}
\]

Using Bekić's theorem [3], we can show that the translation from regular alternation-free \( \mu \)-calculus to PDLR preserves the semantics of formulas: \( [\sigma Y. \varphi] \rho = [(T_1(\sigma Y. \varphi, \sigma), T_2(\sigma Y. \varphi, \varphi))] \rho \) for any context \( \rho : \gamma \rightarrow 2^S \) and \( \sigma \in \{ \mu, \nu \} \). Note also that the size of the PDLR specification obtained is linear in the size of \( \varphi \): there are as many equations in the system as variables in (the expanded form of) \( \varphi \) and as many operators in the right-hand sides as operators in \( \varphi \). However, in order to obtain a succinct translation into Bess, we need simple PDLR specifications, i.e., in which all PDL formulas in right-hand sides contain at most one boolean or modal operator. This is easily done by splitting the PDL formulas and introducing new variables, and may cause at most a linear blow-up in the size of the equation system. For the example above, we obtain the following equivalent simple PDLR specification:

\[
\{ Y_0, Y_0 \overset{\nu}{\leftarrow} \left[T^* . \text{Send}\right] Y_1 \}, \{ Y_1 \overset{\top}{\leftarrow} Y_2 \land Y_3, Y_2 \overset{\top}{\leftarrow} \land \neg \text{Recv} \ Y_1 \}
\]

**Translation into HML with recursion.** The second step is to translate a simple PDLR specification into HMLR, which amounts to eliminate all regular operators inside the modal formulas present in the right-hand sides of the equation system. This translation is performed by the syntactic function \( R \) defined in Figure 5. Every equation containing a modality with a regular expression is translated into (one or more) equations of the same sign that contain modalities with simpler regular formulas (having less regular operators). This process

RR n° 3899
continues recursively until all resulting modalities in the right-hand sides belong to HML, i.e., they contain only pure action formulas.

\[
\mathbf{R}(Y; M_1, \ldots, M_p) = (Y; \mathbf{R}(M_1), \ldots, \mathbf{R}(M_p))
\]

\[
\mathbf{R}([\{ Y_i \models \varphi_i \}_{1 \leq i \leq n}]) = \bigcup_{i=1}^n \mathbf{R}(Y_i \models \varphi_i)
\]

\[
\mathbf{R}(Y \models (\alpha) \varphi) = \{ Y \models \alpha \}
\]

\[
\mathbf{R}(Y \models [\alpha] \varphi) = \{ Y \models [\alpha] \}
\]

\[
\mathbf{R}(Y \models \langle \beta_1, \beta_2 \rangle \varphi) = \mathbf{R}(Y \models \langle \beta_1 \rangle Y_1) \cup \mathbf{R}(Y_1 \models \langle \beta_2 \rangle \varphi)
\]

\[
\mathbf{R}(Y \models [\beta_1, \beta_2] \varphi) = \mathbf{R}(Y \models [\beta_1] Y_1) \cup \mathbf{R}(Y_1 \models [\beta_2] \varphi)
\]

\[
\mathbf{R}(Y \models \langle \beta \rangle \varphi) = \{ Y \models \varphi \lor Y_1 \} \cup \mathbf{R}(Y_1 \models \langle \beta \rangle Y)
\]

\[
\mathbf{R}(Y \models [\beta] \varphi) = \{ Y \models \varphi \land Y_1 \} \cup \mathbf{R}(Y_1 \models [\beta] Y)
\]

**Fig. 5.** Translation of simple PDLR specifications in HMLR

For the simple PDLR specification obtained in the previous example, the translation \( \mathbf{R} \) yields the following (simple) HMLR specification:

\[
(Y_0, \{ Y_0 \models Y_4 \land Y_5, Y_4 \models [Send] Y_1, Y_5 \models [T] Y_0 \}, \{ Y_1 \models Y_2 \land Y_3, Y_2 \models [T] Y_3, Y_3 \models [\neg \text{Rec}] Y_1 \})
\]

The translation from PDLR to HMLR preserves the semantics of specifications: \([ (Y, M_1, \ldots, M_p) ] \rho = [ \mathbf{R}(Y, M_1, \ldots, M_p) ] \rho \) for any context \( \rho : \mathcal{Y} \rightarrow 2^S \). Moreover, it is easy to see that \( \mathbf{R} \) may cause at most a linear blow-up in the size of the equation system.

**Translation into BESs.** The third step is to translate a simple HMLR specification into an (alternation-free) boolean equation system. A BES (see Figure 6) consists of a boolean variable \( x \) and a fixed point equation system \( B_1, \ldots, B_p \) with boolean variables in left-hand sides and boolean formulas in right-hand sides. For simplicity, we consider only pure disjunctive or conjunctive boolean formulas. An empty disjunction is equivalent to \( F \) and an empty conjunction is equivalent to \( T \). The semantics of a BES is defined in a way similar to a PDLR specification, except that it produces the boolean value associated to \( x \) in the solution of \( B_1, \ldots, B_p \).

The local model-checking of a (simple) HMLR specification \((Y, M_1, \ldots, M_p)\) on the initial state \( s_0 \) of an LTS \( L = (S, A, T, s_0) \) means to decide whether the set of states denoted by \( Y \) contains \( s_0 \). This is translated into a BES by the semantic function \( \mathbf{B} \) defined inductively in Figure 7. To every propositional variable \( Y \) in the left-hand side of an equation and to
Efficient Model-Checking for Regular Mu-Calculus

Fig. 6. Syntax and semantics of boolean equation systems

every state \( s \in S \) is associated a boolean variable \( Y_s \) encoding the fact that \( s \) belongs to the set of states denoted by \( Y \). To every HMLt formul\( \varphi \) in a right-hand side and to every state \( s \) is associated a boolean formula \( B(\varphi, s) \) encoding the fact that \( s \) satisfies \( \varphi \).

Fig. 7. Translation of simple HMLR specifications into Bess

The \( B \) function is similar to other translations from modal equation systems to Bess \([2,5,1,29,21]\). \( B \) produces a BES whose size is linear in the size of the HMLR specification (which in turn is linear in the size of the initial state formula) and the size of the LTS (number of states and transitions). It is important to note that during the translation of modal formulas (see Figure 7), the transitions in the LTS are traversed forwards, which enables to construct the LTS in a demand-driven way during the verification.

RR n° 3899
3.2 Local resolution of BESs

The final step of the model-checking procedure is the local resolution of the alternation-free BES obtained by translating the local verification of a formula $\varphi$ on an LTS $(S, A, T, s_0)$. As we saw in Section 3.1, the verification of a fixed point formula $\sigma Y. \varphi$ on the initial state $s_0$ amounts to compute the value of the boolean variable $Y_{s_0}$ contained in the first $\sigma$-block of the BES.

For simplicity, we consider here the resolution of BESs containing a single $\mu$-block (the solving routine for $\nu$-blocks is completely dual). Multiple-block alternation-free BESs can be handled by associating to each $\sigma$-block in the BES its corresponding solving routine. Every $\alpha$ variable $x_j$ bound in a $\sigma$-block $B_j$ is required in another block $B_j$ that depends on $B_j$, the solving routine of $B_j$ is called to compute $x_j$. The computation of $x_j$ may require in turn the values of other variables that are free in $B_j$ and defined in other blocks, leading to calls of the routines corresponding to those blocks, and so on. This process will eventually stop, because the BES being alternation-free, there are no cyclic dependencies between blocks. During the resolution, the same variable of a block may be required several times in other blocks; therefore, the computation results must be persistent between subsequent calls of the same solving routine\(^1\).

**Extended Boolean Graphs.** Our resolution algorithm is easier to develop using a representation of BESs as *extended boolean graphs* [24], which are a slight generalization of the boolean graphs proposed in [1]. An extended boolean graph (EBG) is a tuple $G = (V, E, L, F)$, where: $V$ is the set of vertices; $E \subseteq V \times V$ is the set of edges; $L : V \rightarrow \{v, \land\}$ is the vertex labeling; and $F \subseteq V$ is the *frontier* of $G$. Intuitively, the frontier of an EBG $G$ contains the only vertices of $G$ starting at which new edges can be added when $G$ is embedded in another EBG. The set of successors of a vertex $x \in V$ is noted $E(x)$.

A closed BES can be represented by an EBG $G = (V, E, L, \emptyset)$, where $V$ denotes the set of boolean variables, $E$ denotes the dependencies between variables, and $L$ labels the vertices as disjunctive or conjunctive according to the operator in the corresponding equation of the BES (the frontier set is empty since $G$ is not meant to be embedded in another graph). Figure 8 shows a closed BES and its associated EBG, where black (white) vertices denote variables that are true (false) in the BES solution. The grey area delimits a subgraph containing the vertices $\{x_0, x_3, x_4, x_5, x_8\}$ and having the frontier $\{x_0, x_3, x_8\}$.

Every EBG $G = (V, E, L, F)$ induces a Kripke structure $G = (V, E, L, F)$. Such a Kripke structure is represented in an *implicit* manner when the "successor" function $E(x)$ can be computed for every vertex $x \in V$ without knowing the whole set $V$ (this is the case for the successor function implemented by the translation B given in Figure 7).

Let $P_v$ and $P_h$ be two atomic propositions denoting the $\lor$- and $\land$-vertices of a Kripke structure $G$ induced by a BES. The BES solution can be characterized by the following $\mu$-calculus formula interpreted over $G$ [24]:

$$ \text{Ex} = \mu Y (P_v \land (T) Y) \lor (P_h \land [T] Y) $$

\(^1\) This resolution scheme could be naturally implemented using coroutines.
A variable $x$ of the BES is true iff the vertex $x$ satisfies $\text{Ex}$ in $G$, noted $x \models_G \text{Ex}$. Intuitively, $\text{Ex}$ expresses that some (all) successors of a $\vee$-vertex ($\wedge$-vertex) lead, in a finite number of steps, to vertices corresponding to $T$ variables of the BES (these are $\wedge$-vertices without successors, characterized by the formula $P_\wedge \land [T] F$). For the EBG in Figure 8, it is easy to check that the set $\{x_0, x_3, x_4, x_5, x_8\}$ of black vertices is equal to the interpretation of $\text{Ex}$ on $G$, noted $[\text{Ex}]_G$. Thus, the local resolution of a BES amounts to the local model-checking of the $\text{Ex}$ formula on the corresponding Kripke structure.

Consider an EBG $G = (V, E, L, 0)$, its associated Kripke structure $G = (V, E, L)$, and $x \in V$. The local model-checking of $\text{Ex}$ on $x$ does not always require to entirely explore $G$ (e.g., on Figure 8, one could explore only the outlined subgraph in order to check $\text{Ex}$ on $x_0$), but rather to explore a part $G'$ of $G$ such that the value of $x$ can be computed based only on the information in $G'$. Formally, this means to compute a subgraph $G' = (V', E', L', F')$ of $G$ that contains $x$ and is solution-closed [24], i.e., the satisfaction of $\text{Ex}$ by $x$ is the same in $G'$ and $G$: $[\text{Ex}]_{G'} = [\text{Ex}]_G \cap V'$. A subgraph $G'$ is solution-closed iff the satisfaction of $\text{Ex}$ on its frontier $F'$ can be decided using only the information in $G'$: $F' \subseteq [(P \land \text{Ex}) \lor (P \land \neg \text{Ex})]_G$. For the EBG on Figure 8, it is easy to see that the subgraph outlined is solution-closed: its frontier $\{x_0, x_3, x_8\}$ contains only $\vee$-vertices satisfying $\text{Ex}$.

**Local resolution algorithm.** The Solve algorithm that we propose (see Figure 9) takes as input an implicit Kripke structure $G = (V, E, L)$ induced by an EBG $G$ and a vertex $x \in V$ on which the $\text{Ex}$ formula must be checked. Starting from $x$, Solve performs a depth-first search (DFS) of $G$ and simultaneously checks $\text{Ex}$ on all visited vertices, which are stored in a set $A \subset V$. Upon termination, the subgraph $G_A$ of $G$ containing all vertices in $A$ and all edges traversed during the DFS is solution-closed ($[\text{Ex}]_{G_A} = [\text{Ex}]_G \cap A$), meaning that the truth value of $\text{Ex}$ on $x$ computed in $G_A$ is the same as that in $G$.

Solve is similar in spirit with other graph-based local resolution algorithms like those of Andersen [1] and Vergauwen-Lewi [29]. However, since it implements the DFS iteratively, using an explicit stack and two nested while-loops, we believe that Solve is easier to under-
procedure \texttt{SOLVE} \((x, (V,E,L))\) is \\
\begin{align*}
\text{var} & A, B : 2^V; d : V \rightarrow 2^V; \ c, p : V \rightarrow \text{Nat}; \\
y, z, u, w : V; \ stack \ : V^*; \\
c(x) := \text{if } L(x) = \land \text{then } |E(x)| \text{ else } 1; \\
p(x) := 0; \ d(x) := 0; \\
A := \{x\}; \ stack := \text{push}(x, \text{nil}); \\
\text{while } \ stack \neq \text{nil} \text{ do} \\
y := \text{top}(\text{stack}); \\
\text{if } c(y) = 0 \text{ then} \\
\text{if } d(y) \neq \emptyset \text{ then} \\
B := \{y\}; \\
\text{while } B \neq \emptyset \text{ do} \\
\text{let } u \in B; \ B := B \setminus \{u\}; \\
\text{forall } w \in d(u) \text{ do} \\
\text{if } c(w) > 0 \text{ then} \\
c(w) := c(w) - 1; \\
\text{if } c(w) = 0 \text{ then} \\
B := B \cup \{w\} \\
\text{endif} \\
\text{endif} \\
\text{end} \\
d(u) := \emptyset \\
\text{end} \\
\text{else} \\
\text{stack} := \text{pop}(\text{stack}) \\
\text{endif} \\
\text{elseif } p(y) \leq |E(y)| - 1 \text{ then} \\
z := (E(y))_{p(y)}; \ p(y) := p(y) + 1; \\
\text{if } z \in A \text{ then} \\
d(z) := d(z) \cup \{y\} \\
\text{if } c(z) = 0 \text{ then} \\
\text{stack} := \text{push}(z, \text{stack}) \\
\text{endif} \\
\text{else} \\
c(z) := \text{if } L(z) = \land \text{then } |E(z)| \text{ else } 1 \\
p(z) := 0; \ d(z) := \{y\}; \\
A := A \cup \{z\}; \ stack := \text{push}(z, \text{stack}) \\
\text{endif} \\
\text{else} \\
\text{stack} := \text{pop}(\text{stack}) \\
\text{endif} \\
\text{end} \\
\text{end}
\end{align*}
stand than e.g., Andersen’s algorithm, which uses a while-loop and two mutually recursive functions.

The successors $E(y)$ of every vertex $y \in V$ are assumed to be ordered from $(E(y))_0$ to $(E(y))_{|E(y)|−1}$. For every vertex $y \in A$, a counter $p(y)$ denotes the current successor of $y$ that must be explored. Every time a vertex $y$ such that $y \models_G \text{Ex}$ is encountered on top of the stack (this can be either a “new” $\land$-sink vertex, or an already visited vertex), the Ex formula is reevaluated in $G_A$.

This reevaluation is carried out by the inner while-loop by keeping a work set $B \subseteq A$ containing the vertices $u$ such that $u \models_{G_A} \text{Ex}$ and Ex has not yet been reevaluated on the nodes that depend upon $u$. To keep track of these backward dependencies, to each vertex $y \in A$ we associate the set $d(y) \subseteq A$ containing the currently visited predecessor vertices of $y$ (these vertices directly depend upon $y$ and Ex must be reevaluated on them when Ex becomes true on $y$). To efficiently perform the reevaluation of Ex, we use the counter-based technique introduced in [2,5]: to every vertex $y \in A$, we associate a counter $c(y)$ denoting the least number of successors of $y$ that currently have to satisfy Ex in order to ensure $y \models_{G_A} \text{Ex}$ ($c(y)$ is initialized to 1 for $\lor$-vertices and to $|E(y)|$ for $\land$-vertices). Thus, for every $y \in A$, $y \models_{G_A} \text{Ex}$ iff $c(y) = 0$.

Figure 10 shows the result of executing Solve for the variable $x_0$ and the ForC in Figure 8 (during the DFS, the successors of each vertex are visited as if the right-hand side of the corresponding equation was evaluated from left to right). The subgraph $G_A$ computed by Solve, containing the vertices $\{x_0, x_1, x_2, x_3, x_4, x_5\}$, is solution-closed, because its frontier $\{x_0, x_5\}$ contains only $\lor$-vertices satisfying Ex in $G_A$.

![Diagram](image-url)

Fig. 10. A solution-closed subgraph computed by Solve

During the execution of Solve, the DFS stack repeatedly takes one of the three forms outlined on Figure 11.

In form a), all vertices $y$ pushed on the stack are “unstable” ($c(y) > 0$), meaning that the truth of Ex on $y$ depends on the portion $V \setminus A$ of $G$ that has not been explored yet: so, the DFS must continue. In form b), a vertex $y$ that is “stable” ($c(y) = 0$) has been encountered and pushed on top of the stack, meaning that some vertices depending on $y$ may also

RR n° 3899
become stable; therefore, \textsc{Ex} must be reevaluated in \( G_A \). In form c), this reevaluation has been finished, possibly leading to stabilization of some vertices in \( A \); then, all stable vertices present on the stack will be popped, since no further information is needed for them. The \textsc{DFS} properties ensure that all stable vertices on the stack are adjacent to the top\(^2\), and thus after they are popped the stack takes again the form a).

\textsc{Solve} has a linear-time worst-case complexity, since every edge in \( G_A \) is traversed at most twice: forwards (when its target vertex is visited by the \textsc{DFS}) and backwards (when \textsc{Ex} is reevaluated on its source vertex). Moreover, \textsc{Solve} has also a good average-case complexity, improving on Andersen and Vergauwen-Lewi's algorithms, since it stops as soon as \( x \models_{G_A} \text{Ex} \) and explores only vertices that are likely to influence \( x \). Also, backward dependencies \( d(u) \) of stable vertices \( u \) are freed during the inner while-loop, thus reducing memory consumption.

**Diagnostic generation.** Practical applications of \textsc{Bes} resolution, such as temporal logic model-checking, often require a more detailed feedback than a simple yes/no answer. To allow an efficient debugging of the temporal formulas, it is desirable to have also diagnostic information explaining the truth value obtained for the boolean variable of interest. Both positive diagnostics (examples) and negative diagnostics (counterexamples) are needed in order to have a full explanation of a temporal formula.

Let \( G = (V, E, L, F) \) be an \textsc{Eng} and \( x \in V \) the variable of interest. A diagnostic for \( x \) is a solution-closed subgraph \( G' \) of \( G \) that contains \( x \) and is minimal w.r.t. to subgraph inclusion, i.e., it contains the minimal amount of information needed in order to decide the satisfaction of \textsc{Ex} by \( x \). A diagnostic \( G' \) is called example if \( x \models_{G'} \text{Ex} \) and counterexample if \( x \not\models_{G'} \text{Ex} \).

\(^2\) The reevaluation of \textsc{Ex}, which involves a backwards traversal of edges in \( G_A \), can affect only those vertices in the \textsc{DFS} tree that are descendants of stable vertices present on the stack, outlined by the grey portion on Figure 11 c).
The \textsc{Solve} algorithm does not directly produce diagnostics; however, it can be easily coupled with the diagnostic generation algorithms proposed in [24]. These algorithms take as input a solution-closed subgraph (in which the semantics of \textsc{Ex} has been already computed) and construct a diagnostic for a given variable by performing efficient traversals of the subgraph. Figure 12 shows an example for the variable $x_0$ obtained by traversing again the solution-closed subgraph on Figure 10 previously computed by \textsc{Solve}.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure12}
\caption{An example for $x_0$}
\end{figure}

Since these diagnostic generation algorithms have a linear complexity in the size of the solution-closed subgraph they are executed upon, they affect neither the worst-case, nor the average-case complexity of \textsc{Solve}.

4 Implementation and use

We used the model-checking method presented in Section 3 as a basis for developing the \textsc{Evaluator} 3.0 model-checker within the \textsc{Cadp} (\textsc{Caesar/Aldebaran}) toolset [9]. The tool has been built using the \textsc{Open/Caesar} environment [13], which provides a generic API for on-the-fly exploration of transition systems. As a consequence, \textsc{Evaluator} 3.0 can be used in conjunction with every compiler that is \textsc{Open/Caesar}-compliant (i.e., that implements a translation from its input language to the \textsc{Open/Caesar} API), and particularly with the \textsc{Caesar} compiler [14] for \textsc{Lotos}.

4.1 Additional operators and property patterns

Practical experience in using model-checking has shown the need for abstraction mechanisms enabling the specifier to define and use his own temporal operators in addition to those predefined in the model-checker. The input language of \textsc{Evaluator} 3.0 offers a macro-expansion mechanism allowing to define parameterized formulas and an inclusion mechanism allowing to group these definitions into separate libraries that can be reused in temporal specifications.
An immediate application was to build libraries for particular logics like CTL or ACTL by translating their temporal operators as fixed point formulas in regular alternation-free $\mu$-calculus. For example, the $E_t[\varphi_1 \cup \alpha_1, \varphi_2]$ operator of ACTL (stating the existence of a sequence $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots s_k \xrightarrow{a_k} s_{k+1}$ such that $s_i \models \varphi_1$ for all $1 \leq i \leq k$, $a_j \models \alpha_1$ for all $1 \leq j < k$, $a_k \models \alpha_2$, and $s_{k+1} \models \varphi_2$) can be encoded as a macro $\text{EU}_A(\varphi_1, \alpha_1, \alpha_2, \varphi_2) = \mu Y(\varphi_1 \land (\langle \alpha_2 \rangle \varphi_2 \lor \langle \alpha_1 \rangle Y))$. Of course, these particular operators can be freely mixed with the built-in ones in temporal formulas, thus providing added flexibility to advanced users.

Another source of flexibility is provided by the use of wildcards (regular expressions on character strings) instead of action names in the formulas. If transition labels are represented as character strings (as it is currently the case with the OPEN/CESAI API), this allows to specify a set of labels using a single action predicate. For example, the wildcard ‘SEND,*’ represents all transition labels denoting communication of 0 or more values on gate SEND.

In practice, it appears that in many cases, temporal properties tend to belong to particular classes of high-level “property patterns”, such as absence, existence, universality, precedence, and response. These patterns have been identified in [6] after an important statistical study concerning over 500 applications of temporal logic model-checking. The knowledge embedded in this pattern system is important for both expert and non-expert users, since it reduces the risk of specification errors and facilitates the learning of temporal logic-based formalisms.

These property patterns have been expressed in [6] using several specification formalisms (CTL, LTL, regular expressions, etc.) but none of them was directly applicable to description languages with action-based semantics such as process algebras. Therefore, we developed in EVALUATOR 3.0 a library of parameterized formulas implementing the property patterns in regular alternation-free $\mu$-calculus. It turned out that many of them could be expressed in a much more concise and readable form than with the other formalisms used in [6]. Table 2 shows the first three patterns contained in the library.

Besides facilitating the user task at the specification level, it is also important to offer enough feedback on the verification results to allow an easy debugging of the applications. This is achieved through the diagnostic generation facilities provided by EVALUATOR 3.0, which allows to produce examples and counterexamples explaining the truth value of regular alternation-free $\mu$-calculus formulas. As a side effect, this enables the user to get full diagnostics for particular temporal logics implemented as libraries, such as CTL and ACTL. Moreover, EVALUATOR 3.0 can be used to search regular execution sequences in LTL, by checking PDL basic modalities: a transition sequence starting at the initial state and satisfying a regular formula $\beta$ can be obtained either as an example for the $\langle \beta \rangle T$ formula, or as a counterexample for the $[\beta] F$ formula.
Table 2. Property patterns in regular alternation-free $\mu$-calculus

<table>
<thead>
<tr>
<th>Pattern ( $\alpha_1$ is false)</th>
<th>Scope</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absence $\alpha_1$ is false)</td>
<td>Globally</td>
<td>$[T^*.$ $\alpha_1] F$</td>
</tr>
<tr>
<td></td>
<td>Before $\alpha_2$</td>
<td>$[\neg \alpha_2]^<em>.$ $\alpha_1.$ $T^</em>.$ $\alpha_2] F$</td>
</tr>
<tr>
<td></td>
<td>After $\alpha_2$</td>
<td>$[\neg \alpha_2]^<em>.$ $\alpha_2.$ $T^</em>.$ $\alpha_1] F$</td>
</tr>
<tr>
<td></td>
<td>Between $\alpha_2$ and $\alpha_3$</td>
<td>$[T^<em>.$ $\alpha_2.$ $\neg \alpha_3]^</em>.$ $\alpha_1.$ $T^*.$ $\alpha_3] F$</td>
</tr>
<tr>
<td></td>
<td>After $\alpha_2$ until $\alpha_3$</td>
<td>$[T^<em>.$ $\alpha_2.$ $\neg \alpha_3]^</em>.$ $\alpha_1] F$</td>
</tr>
<tr>
<td>Existence ( $\alpha_1$ becomes true)</td>
<td>Globally</td>
<td>$\mu Y.$ $(T) \land \neg \alpha_1] Y$</td>
</tr>
<tr>
<td></td>
<td>Before $\alpha_2$</td>
<td>$[\neg \alpha_1]^*.$ $\alpha_2] F$</td>
</tr>
<tr>
<td></td>
<td>After $\alpha_2$</td>
<td>$[\neg \alpha_2]^*.$ $\alpha_2.$ $\mu Y.$ $(T) \land \neg \alpha_1] Y$</td>
</tr>
<tr>
<td></td>
<td>Between $\alpha_2$ and $\alpha_3$</td>
<td>$[T^<em>.$ $\alpha_2.$ $\neg \alpha_1]^</em>.$ $\alpha_3] F$</td>
</tr>
</tbody>
</table>
|                                | After $\alpha_2$ until $\alpha_3$ | $[T^*.$ $\alpha_2] \left( [\neg \alpha_1]^*.$ $\alpha_3] F \land 
\mu Y.$ $(T) \land \neg \alpha_1] Y\right) F$ |
| Universality ( $\alpha_1$ is true) | Globally | $[T^*.$ $\neg \alpha_1] F$ |
|                                | Before $\alpha_2$ | $[\neg \alpha_2]^*.$ $\neg \alpha_1 \lor \alpha_2.$ $\neg \alpha_2]^*.$ $\alpha_2] F$ |
|                                | After $\alpha_2$ | $[\neg \alpha_2]^*.$ $\alpha_2.$ $\neg \alpha_1] F$ |
|                                | Between $\alpha_2$ and $\alpha_3$ | $[T^*.$ $\alpha_2.$ $\neg \alpha_3]^*.$ $\neg \alpha_1 \lor \alpha_3.$ $T^*.$ $\alpha_3] F$ |
|                                | After $\alpha_2$ until $\alpha_3$ | $[T^*.$ $\alpha_2.$ $\neg \alpha_3]^*.$ $\neg \alpha_1 \lor \alpha_3)] F$ |
4.2 Experimental results

We illustrate below the behaviour of EVALUATOR 3.0 by means of a simple benchmark example: the Alternating Bit Protocol (ABP for short) described in LOTOS. The protocol specification (available in the CADP release) contains four parallel processes: a sender entity, a receiver entity, and two channels modelling the communication of messages and acknowledgements, respectively. The sender accepts messages from a local user through a gate Put and the receiver delivers the messages to a remote user through a gate Get. Messages are represented by natural numbers between 0 and n, where n is a parameter of the specification.

We formulated and verified several safety, liveness, and fairness properties of the ABP (see Table 3). For each property, the table gives its informal meaning, its corresponding regular alternation-free µ-calculus formula, and its truth value on the LOTOS specification. Action predicates Put and Get denote the communication of message i on gates Put and Get, respectively. Predicates Putany and Getany (wildcards) denote the communication of arbitrary messages on gates Put and Get. Every property containing an occurrence of Puti and/or Geti has been checked for all values of i between 0 and n.

<table>
<thead>
<tr>
<th>NO.</th>
<th>PROPERTY</th>
<th>FORMULA</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Initially, a Put will be eventually reached</td>
<td>µY. (T) T \land [\neg Put_{any}] Y</td>
<td>false</td>
</tr>
<tr>
<td>P2</td>
<td>Initially, a Put will be fairly reached</td>
<td>(\neg Put_{any})^* (T^* Put_{any}) T</td>
<td>true</td>
</tr>
<tr>
<td>P3</td>
<td>Initially, no Get can be reached before the corresponding Put</td>
<td>(\neg Get_i)^* F</td>
<td>true</td>
</tr>
<tr>
<td>P4</td>
<td>Between two consecutive Puts, there is a corresponding Get</td>
<td>[T^* Put_i. (\neg Get_i)^* Put_{any}] F</td>
<td>true</td>
</tr>
<tr>
<td>P5</td>
<td>Between two consecutive Gets, there is a corresponding Put</td>
<td>[T^* Get_{any}. (\neg Put_i)^* Get_i] F</td>
<td>true</td>
</tr>
<tr>
<td>P6</td>
<td>After a Put, the corresponding Get is eventually reachable</td>
<td>[T^* Put_i. \mu Y. (T) T \land [\neg Get_i] Y</td>
<td>false</td>
</tr>
<tr>
<td>P7</td>
<td>After a Put, the corresponding Get is fairly reachable</td>
<td>[T^* Put_i. (\neg Get_i)^* (\neg Get_i)^* Get_i] T</td>
<td>true</td>
</tr>
</tbody>
</table>

Properties $P_1$ and $P_6$, which express the inevitable reachability of Put and Get actions, are false because of the livelocks ($\tau$-loops) present in the LOTOS description. These two properties can be reformulated — as $P_2$ and $P_7$, respectively — in order to state the inevitable reachability only over fair execution sequences (i.e., by skipping loops).

We performed several experiments with EVALUATOR 3.0, by checking all properties on the ABP specification for different values of n. For comparison, we also used the
Evaluator 2.0 model-checker developed at VERIMAG, which accepts as input plain alternation-free $\mu$-calculus formulas and implements the Fernandez-Mounier local boolean resolution algorithm [11]. All experiments have been performed on a Sparc Ultra 1 machine with 256 Mbytes of memory.

The results are shown in Table 4. For each experiment, the table gives the number of states of the LTS, the time (in minutes) required for the local model-checking of each property, and the percentage of states explored by each tool. The Solve algorithm performs uniformly better than the Fernandez-Mounier algorithm, the time needed being at least 50% smaller and the percentage of LTS states explored being always smaller or equal. For properties $P_1$, $P_2$, and $P_5$, which require to explore only a very small part of the LTS in order to decide their truth value, Evaluator 3.0 stops almost instantaneously (less than a second) in all cases, while Evaluator 2.0 takes up to one hour for $n = 100$.

**Table 4. Local model-checking statistics**

<table>
<thead>
<tr>
<th>No.</th>
<th>$n = 20$</th>
<th>$n = 40$</th>
<th>$n = 60$</th>
<th>$n = 80$</th>
<th>$n = 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>S</td>
<td>= 30,800$</td>
<td>$</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>expl. %</td>
<td>time</td>
<td>expl. %</td>
<td>time</td>
</tr>
<tr>
<td>$P_1$</td>
<td>a</td>
<td>0'</td>
<td>0.01</td>
<td>0'</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>20'</td>
<td>93.1%</td>
<td>1'42'</td>
<td>96.4%</td>
</tr>
<tr>
<td>$P_2$</td>
<td>a</td>
<td>0'</td>
<td>0.01</td>
<td>0'</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>1'02'</td>
<td>100%</td>
<td>5'11'</td>
<td>100%</td>
</tr>
<tr>
<td>$P_3$</td>
<td>a</td>
<td>8'</td>
<td>91.7%</td>
<td>37'</td>
<td>95.7%</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>16'</td>
<td>91.7%</td>
<td>1'09'</td>
<td>95.7%</td>
</tr>
<tr>
<td>$P_4$</td>
<td>a</td>
<td>9'</td>
<td>100%</td>
<td>37'</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>19'</td>
<td>100%</td>
<td>1'14'</td>
<td>100%</td>
</tr>
<tr>
<td>$P_5$</td>
<td>a</td>
<td>18'</td>
<td>100%</td>
<td>1'15'</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>38'</td>
<td>100%</td>
<td>3'01'</td>
<td>100%</td>
</tr>
<tr>
<td>$P_6$</td>
<td>a</td>
<td>0'</td>
<td>0.02</td>
<td>0'</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>48'</td>
<td>100%</td>
<td>3'34'</td>
<td>100%</td>
</tr>
<tr>
<td>$P_7$</td>
<td>a</td>
<td>10'</td>
<td>100%</td>
<td>38'</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>20'</td>
<td>100%</td>
<td>1'18'</td>
<td>100%</td>
</tr>
</tbody>
</table>

(a) Evaluator 3.0 (Solve algorithm)
(b) Evaluator 2.0 (Fernandez-Mounier algorithm)
5 Conclusion and future work

We presented an efficient method for on-the-fly model-checking of regular alternation-free \( \mu \)-calculus formulas over finite labeled transition systems. The method is based on a succinct reduction of the verification problem to a boolean equation system, which is solved using an efficient local algorithm. Used in conjunction with specialized diagnostic generation algorithms [24], the method also allows to produce examples and counterexamples fully explaining the truth values of the formulas. The method has been implemented in the model-checker EVALUATOR 3.0 that we developed as part of the CADP (CÆSAR/ALDEBARAN) protocol engineering toolset [9] using the OPEN/CÆSAR environment [13].

The input language of EVALUATOR 3.0 allows to define reusable libraries containing new temporal logic operators expressed in regular alternation-free \( \mu \)-calculus. At the present time, we developed libraries encoding the operators of \( \text{CTL} \) [4], \( \text{ACTL} \) [25], and a collection of generic property patterns proposed in [6] intended to facilitate the temporal logic specification activity.

EVALUATOR 3.0 has been successfully experimented on various specifications of communication protocols and distributed applications (see for instance the examples in the CADP release). The diagnostic generation features and the possibility of defining separate libraries of temporal operators appeared to be extremely useful in practice. Moreover, a connection between EVALUATOR 3.0 and the ORCCAD environment for robot controller design [26], including a graphical interface for the property pattern system, is currently under development.

In the future, we plan to apply EVALUATOR 3.0 also for bisimulation/predorder checking, by using the characteristic formula approach [16] that allows to compare two labeled transition systems \( M_1 \) and \( M_2 \) by constructing a characteristic formula of \( M_1 \) and verifying it on \( M_2 \). Also, the diagnostic generation features could be useful in the framework of test generation based on verification [10]. Using again the characteristic formula approach, test purposes could be described as temporal formulas and the corresponding test cases would be obtained as diagnostics for these formulas.

Finally, we plan to extend the logic of EVALUATOR 3.0 with data variables, which allow to reason more naturally about systems described in value-passing process algebras such as \( \mu \text{CRL} \) [15] and full LOTOS [17]. This can be done by translating data-based temporal logic formulas into parameterized boolean equation systems, which can be solved on-the-fly [23]. The implementation of these algorithms within the CADP toolset will require the extension of the OPEN/CÆSAR environment with data-handling facilities.

Acknowledgements

This work was partially supported by the INRIA Cooperative Research Action TOLERE directed by Alain Girault. We are also grateful to Hubert Garavel for his useful comments and for providing valuable assistance during the development of the EVALUATOR 3.0 model-checker. Versions 1.0 and 2.0 of EVALUATOR [11] were developed by Marius Bozga and Laurent Mounier from VERIMAG.
References


RR n° 3899


