INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Radu Mateescu — Mihaela Sighireanu

N° 3899

Mars 2000

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--3899--FR+ENG

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 μ-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

This report is also available as "Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus," in Ina Schieferdecker and Axel Rennoch, editors, Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany), ERCIM, April 2000.

* Radu.Mateescu@inria.fr
** Mihaela.Sighireanu@liafa.jussieu.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

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 μ-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-Δ [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 μ-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 μ-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 (Cæsar/Aldébaran) toolset [9] using the generic Open/Cæsar 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 implementation 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-Δ [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 $L = (S, A, T, s_0)$, where: S is a finite set of *states*, A is a finite set of *actions*, $T \subseteq S \times A \times S$ is the *transition relation*, and $s_0 \in S$ is the *initial state*. A transition $(s, a, s') \in T$, also noted $s \xrightarrow{a} s'$, 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	$\alpha ::= a \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2$
Regular formulas	$\beta ::= \alpha \mid \beta_1.\beta_2 \mid \beta_1 \vert \beta_2 \mid \beta^*$
State formulas	$\varphi ::= \mathsf{F} \mid \mathsf{T} \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle\beta\rangle\,\varphi \mid [\beta]\,\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi$

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

Action formulas α are built from action names $a \in A$ by using the standard boolean operators. Derived boolean connectives are defined as usual: $\mathsf{F} = a \wedge \neg a$ for some a, $\mathsf{T} = \neg\mathsf{F}$, $\alpha_1 \vee \alpha_2 = \neg(\neg\alpha_1 \wedge \neg\alpha_2)$, etc. Regular formulas β are built from action formulas α by using the standard regular expression operators: concatenation (.), choice (|), and transitive-reflexive closure ($*$). The empty sequence operator ε and the transitive closure operator $+$ are defined as $\varepsilon = \mathsf{F}^*$ and $\beta^+ = \beta.\beta^*$. State formulas φ are built from propositional variables $Y \in \mathcal{Y}$ by using the standard boolean operators, the possibility and necessity operators $\langle\beta\rangle\,\varphi$ and $[\beta]\,\varphi$, and the minimal and maximal fixed point operators $\mu Y.\varphi$ and $\nu Y.\varphi$. The μ and ν operators act as binders for Y variables in a way similar to quantifiers in first-order logic. A formula φ without free occurrences of Y variables is *closed*. Formulas φ are assumed to be *alternation-free*, i.e., without mutually recursive minimal and maximal fixed point subformulas ($\langle\beta\rangle\,\varphi'$ and $[\beta]\,\varphi'$ modalities, where β contains $*$ operators, must be considered as "hidden" minimal and maximal fixed point subformulas, respectively).

Action formulas

$$\begin{aligned}
[\![a]\!] &= \{a\} \\
[\![\neg\alpha]\!] &= A \setminus [\![\alpha]\!] \\
[\![\alpha_1 \wedge \alpha_2]\!] &= [\![\alpha_1]\!] \cap [\![\alpha_2]\!]
\end{aligned}$$

Regular formulas

$$\begin{aligned}
\|\alpha\| &= \{(s, s') \in S \times S \mid \exists a \in A. s \xrightarrow{a} s' \wedge a \in [\![\alpha]\!]\} \\
\|\beta_1.\beta_2\| &= \|\beta_1\| \circ \|\beta_2\| \\
\|\beta_1|\beta_2\| &= \|\beta_1\| \cup \|\beta_2\| \\
\|\beta^*\| &= \|\beta\|^*
\end{aligned}$$

State formulas

$$\begin{aligned}
[\![\mathsf{F}]\!]\,\rho &= \emptyset \\
[\![\mathsf{T}]\!]\,\rho &= S \\
[\![\varphi_1 \vee \varphi_2]\!]\,\rho &= [\![\varphi_1]\!]\,\rho \cup [\![\varphi_2]\!]\,\rho \\
[\![\varphi_1 \wedge \varphi_2]\!]\,\rho &= [\![\varphi_1]\!]\,\rho \cap [\![\varphi_2]\!]\,\rho \\
[\![\langle\beta\rangle\,\varphi]\!]\,\rho &= \{s \in S \mid \exists s' \in S.(s, s') \in \|\beta\| \wedge s' \in [\![\varphi]\!]\,\rho\} \\
[\![[\beta]\,\varphi]\!]\,\rho &= \{s \in S \mid \forall s' \in S.(s, s') \in \|\beta\| \Rightarrow s' \in [\![\varphi]\!]\,\rho\} \\
[\![Y]\!]\,\rho &= \rho(Y) \\
[\![\mu Y.\varphi]\!]\,\rho &= \bigcap\{S' \subseteq S \mid \Phi_\rho(S') \subseteq S'\} \\
[\![\nu Y.\varphi]\!]\,\rho &= \bigcup\{S' \subseteq S \mid S' \subseteq \Phi_\rho(S')\}
\end{aligned}$$

where $\Phi_\rho : 2^S \to 2^S$, $\Phi_\rho(S') = [\![\varphi]\!]\,(\rho \oslash [S'/Y])$

Fig. 2. Semantics of regular alternation-free μ-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 α. The interpretation $\|\beta\| \subseteq S \times S$ of regular formulas gives a binary relation between the source and target states of transition sequences satisfying β ($\circ$, $\cup$, and $*$ denote composition, union, and transitive-reflexive closure of binary relations). The α regular formula characterizes one-step sequences $s \xrightarrow{a} s'$ such that a satisfies α. The $\beta_1.\beta_2$ formula states that a sequence is the concatenation of two sequences satisfying β_1 and β_2; $\beta_1|\beta_2$ states that a sequence can satisfy β_1 or β_2; and β^* states that a sequence is the concatenation of (zero or more) sequences satisfying β. The interpretation $[\![\varphi]\!]\,\rho \subseteq S$ of state formulas, where the propositional context $\rho : \mathcal{Y} \to 2^S$ assigns state sets to propositional variables, gives the set of LTS states satisfying φ in the context of ρ ($\oslash$ denotes context overriding). The modalities $\langle\beta\rangle\,\varphi$ and $[\beta]\,\varphi$ characterize the states for which some (all) outgoing transition sequences satisfying β lead to states satisfying φ. 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 α (written as $a \models \alpha$) iff $a \in [\![\alpha]\!]$. A state $s \in S$ satisfies a closed formula φ (written $s \models \varphi$) iff $s \in [\![\varphi]\!]$. L is a φ-model (written $L \models \varphi$) iff $[\![\varphi]\!] = S$. 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 $[\mathsf{T}^*]\,\varphi$ (equivalent to the ACTL formula $\mathsf{AG}_{\mathsf{T}}\varphi$), stating that φ holds on every state reachable from s_0.

2.2 Examples

The regular alternation-free μ-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 1. Examples of properties in regular alternation-free μ-calculus

CLASS	PROPERTY	FORMULA
Safety	Absence of `Error` actions	$[\mathsf{T}^*.\mathtt{Error}]\,\mathsf{F}$
	Unreachability of a `Recv` action before a `Send`	$[(\neg\mathtt{Send})^*.\mathtt{Recv}]\,\mathsf{F}$
	Mutual exclusion of sections delimited by `Open` and `Close`	$[\mathsf{T}^*.\mathtt{Open1}.(\neg\mathtt{Close1})^*.\mathtt{Open2}]\,\mathsf{F}$
Liveness	Deadlock freedom: absence of states without successors	$[\mathsf{T}^*]\,\langle\mathsf{T}\rangle\,\mathsf{T}$
	Potential reachability (via some `Errors`) of a `Recv` after a `Send`	$\langle\mathsf{T}^*.\mathtt{Send}.(\mathsf{T}^*.\mathtt{Error})^*.\mathtt{Recv}\rangle\,\mathsf{T}$
	Inevitable reachability of a `Grant` action after a `Request`	$[\mathsf{T}^*.\mathtt{Request}]\,\mu Y.\,\langle\mathsf{T}\rangle\,\mathsf{T} \wedge [\neg\mathtt{Grant}]\,Y$
Fairness	Livelock freedom: absence of `tau`-circuits	$[\mathsf{T}^*]\,\mu Y.\,[\mathtt{tau}]\,Y$
	Fair reachability (by skipping circuits) of a `Recv` after a `Send`	$[\mathsf{T}^*.\mathtt{Send}.(\neg\mathtt{Recv})^*]\,\langle(\neg\mathtt{Recv})^*.\mathtt{Recv}\rangle\,\mathsf{T}$

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 μ-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 $\mu Y_1.(\langle \texttt{Send} \rangle\, \mu Y_2.(\langle \texttt{Recv} \rangle\, \mathsf{T} \vee \mu Y_3.(\langle \texttt{Error} \rangle\, Y_2 \vee \langle \mathsf{T} \rangle\, Y_3)) \vee \langle \mathsf{T} \rangle\, Y_1)$.

Other, more elaborate examples of generic temporal properties encoded in regular alternation-free μ-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 μ-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 φ 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 μ-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 μ-calculus, one way to proceed could be first to translate a state formula φ in plain alternation-free μ-calculus and then to apply the above procedure. This means to encode the regular modalities of φ using fixed point operators, e.g., by applying the Emerson-Lei translation from PDL to alternation-free μ-calculus [7]. This translation is succinct (it produces at most a linear blow-up in the size of φ), 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 μ-calculus formula φ into PDL *with recursion* (PDLR), which is a generalization of the Hennessy-Milner logic with recursion 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 $M_1 . \dots . M_p$ of σ-blocks ($.$ 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 σ-block M_j (for $1 \le j < p$) 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 σ-blocks $M_1, \ldots, M_p$ (usually in M_1). A PdlR specification is *closed* if all variables occurring in it are bound in the equation system.

Syntax of a PdlR specification:

$$P = (Y, M_1 \ldotp \ldots \ldotp M_p)$$

where $M_j = \{Y_{j_i} \stackrel{\sigma_j}{=} \varphi_{j_i}\}_{1 \le i \le n_j}$ for all $1 \le j \le p$

Semantics w.r.t. an Lts (S, A, T, s_0) and a context $\rho : \mathcal{Y} \to 2^S$:

$$[\![(Y, M_1 \ldotp \ldots \ldotp M_p)]\!]\, \rho = (\rho \oslash [\![M_1 \ldotp \ldots \ldotp M_p]\!]\, \rho)(Y)$$

$$[\![M_j \ldotp \ldots \ldotp M_p]\!]\, \rho = ([\![M_j]\!]\, (\rho \oslash [\![M_{j+1} \ldotp \ldots \ldotp M_p]\!]\, \rho)) \ldotp [\![M_{j+1} \ldotp \ldots \ldotp M_p]\!]\, \rho$$

$$[\![\{Y_{j_i} \stackrel{\sigma_j}{=} \varphi_{j_i}\}_{1 \le i \le n_j}]\!]\, \rho = [\sigma_j \overline{\Phi}_{j_\rho} / (Y_{j_1}, \ldots, Y_{j_{n_j}})]$$

where $\overline{\Phi}_{j_\rho} : (2^S)^{n_j} \to (2^S)^{n_j}$, $\overline{\Phi}_{j_\rho}(U_1, ..., U_{nj}) = ([\![\varphi_i]\!]\, (\rho \oslash [U_1/Y_1, ..., U_{nj}/Y_{nj}]))_{1 \le i \le n_j}$

Fig. 3. Syntax and semantics of PdlR

A PdlR specification $(Y, M_1 \ldotp \ldots \ldotp M_p)$ interpreted over an Lts yields the set of states associated to Y in the solution of $M_1 \ldotp \ldots \ldotp M_p$. The solution of $M_1 \ldotp \ldots \ldotp M_p$ is a propositional context in $\mathcal{Y} \to 2^S$ obtained by concatenating the solutions of all σ-blocks M_j $(1 \le j < p)$, each one being calculated in the context of the subsystem $M_{j+1} \ldotp \ldots \ldotp M_p$. The solution of a σ-block M_j with n_j variables is a context mapping M_j's variables to the σ_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 μ-calculus formula φ in PdlR, we must convert φ into *expanded* form, by performing two actions: (a) add a new μY (νY) operator, where Y is a "fresh" variable, in front of every $\langle \beta \rangle\, \varphi_1$ $([\beta]\, \varphi_1)$ subformula of φ in which β contains a $*$ operator (recall from Section 2.1 that these modalities are considered as "hidden" fixed point operators); (b) if the resulting formula φ_0 is not a fixed point one, add in front of φ_0 a σY_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 $(\mathbf{T_1}(\sigma Y_0.\varphi_0, \sigma), \mathbf{T_2}(\sigma Y_0.\varphi_0, \sigma))$ is obtained using two syntactic functions $\mathbf{T_1}$ and $\mathbf{T_2}$, defined inductively in Figure 4. $\mathbf{T_1}(\varphi, \sigma)$ yields a formula obtained from φ by substituting each fixed point subformula by its corresponding variable. $\mathbf{T_2}(\varphi, \sigma)$ yields a system containing, for each fixed point subformula of φ, an equation with the corresponding variable in the left-hand side and a Pdl formula in the right-hand side. The first σ-block, denoted by $hd(\mathbf{T_2}(\varphi, \sigma))$, contains the equations of sign σ associated to the topmost fixed point subformulas of φ. The remainder of the system, denoted by $tl(\mathbf{T_2}(\varphi, \sigma))$, contains the σ-blocks already constructed from subformulas of φ. A new σ-block is created every time that a fixed point subformula with a sign $\tilde{\sigma}$ dual to σ is encountered ($\tilde{\mu} = \nu$ and $\tilde{\nu} = \mu$).

<table>
<tr><th>φ</th><th>$\mathbf{T_1}(\varphi, \sigma)$</th><th>$\mathbf{T_2}(\varphi, \sigma)$</th></tr>
<tr><td>F</td><td>F</td><td rowspan="2">$\{\,\}$</td></tr>
<tr><td>T</td><td>T</td></tr>
<tr><td>$\langle\beta\rangle\, \varphi_1$</td><td>$\langle\beta\rangle\, \mathbf{T_1}(\varphi_1, \sigma)$</td><td rowspan="2">$\mathbf{T_2}(\varphi_1, \sigma)$</td></tr>
<tr><td>$[\beta]\, \varphi_1$</td><td>$[\beta]\, \mathbf{T_1}(\varphi_1, \sigma)$</td></tr>
<tr><td>$\varphi_1 \vee \varphi_2$</td><td>$\mathbf{T_1}(\varphi_1, \sigma) \vee \mathbf{T_1}(\varphi_2, \sigma)$</td><td rowspan="2">$(hd(\mathbf{T_2}(\varphi_1, \sigma)) \cup hd(\mathbf{T_2}(\varphi_2, \sigma)))\mathbf{.}$
$tl(\mathbf{T_2}(\varphi_1, \sigma))\mathbf{.}tl(\mathbf{T_2}(\varphi_2, \sigma))$</td></tr>
<tr><td>$\varphi_1 \wedge \varphi_2$</td><td>$\mathbf{T_1}(\varphi_1, \sigma) \wedge \mathbf{T_1}(\varphi_2, \sigma)$</td></tr>
<tr><td>Y</td><td rowspan="3">Y</td><td>$\{\,\}$</td></tr>
<tr><td>$\sigma Y.\varphi_1$</td><td>$(\{Y \stackrel{\sigma}{=} \mathbf{T_1}(\varphi_1, \sigma)\} \cup hd(\mathbf{T_2}(\varphi_1, \sigma)))\mathbf{.}tl(\mathbf{T_2}(\varphi_1, \sigma))$</td></tr>
<tr><td>$\tilde{\sigma} Y.\varphi_1$</td><td>$\{\,\}\mathbf{.}(\{Y \stackrel{\tilde{\sigma}}{=} \mathbf{T_1}(\varphi_1, \tilde{\sigma})\} \cup hd(\mathbf{T_2}(\varphi_1, \tilde{\sigma})))\mathbf{.}tl(\mathbf{T_2}(\varphi_1, \tilde{\sigma}))$</td></tr>
</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 `Send` action in the Lts will be eventually followed by a `Recv`:

$$\varphi = \nu Y_0.\, [\mathsf{T}^*.\mathtt{Send}]\, \mu Y_1.\, \langle \mathsf{T} \rangle\, \mathsf{T} \wedge [\neg \mathtt{Recv}]\, Y_1$$

The translation $(\mathbf{T_1}(\varphi, \nu), \mathbf{T_2}(\varphi, \nu))$ yields the PdlR specification below:

$$\big(Y_0, \{Y_0 \stackrel{\nu}{=} [\mathsf{T}^*.\mathtt{Send}]\, Y_1\}\mathbf{.}\{Y_1 \stackrel{\mu}{=} \langle \mathsf{T} \rangle\, \mathsf{T} \wedge [\neg \mathtt{Recv}]\, Y_1\}\big)$$

Using Bekić's theorem [3], we can show that the translation from regular alternation-free μ-calculus to PdlR preserves the semantics of formulas: $[\![\sigma Y.\varphi]\!]\, \rho = [\![(\mathbf{T_1}(\sigma Y.\varphi, \sigma), \mathbf{T_2}(\sigma Y.\varphi, \sigma))]\!]\, \rho$ for any context $\rho : \mathcal{Y} \to 2^S$ and $\sigma \in \{\mu, \nu\}$. Note also that the size of the PdlR specification obtained is linear in the size of φ: there are as many equations in the system as variables in (the expanded form of) φ and as many operators in the right-hand sides as operators in φ. 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:

$$\big(Y_0, \{Y_0 \stackrel{\nu}{=} [\mathsf{T}^*.\mathtt{Send}]\, Y_1\}\mathbf{.}\{Y_1 \stackrel{\mu}{=} Y_2 \wedge Y_3, Y_2 \stackrel{\mu}{=} \langle \mathsf{T} \rangle\, \mathsf{T}, Y_3 \stackrel{\mu}{=} [\neg \mathtt{Recv}]\, Y_1\}\big)$$

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

$$
\begin{array}{rcl}
\mathbf{R}(Y, M_1.\ldots.M_p) & = & (Y, \mathbf{R}(M_1).\ldots.\mathbf{R}(M_p)) \\[1em]
\mathbf{R}(\{Y_i \stackrel{\sigma}{=} \varphi_i\}_{1 \le i \le n}) & = & \bigcup_{i=1}^{n} \mathbf{R}(Y_i \stackrel{\sigma}{=} \varphi_i) \\[1em]
\mathbf{R}(Y \stackrel{\sigma}{=} \langle \alpha \rangle \varphi) & = & \{Y \stackrel{\sigma}{=} \langle \alpha \rangle \varphi\} \\
\mathbf{R}(Y \stackrel{\sigma}{=} [\alpha] \varphi) & = & \{Y \stackrel{\sigma}{=} [\alpha] \varphi\} \\
\mathbf{R}(Y \stackrel{\sigma}{=} \langle \beta_1.\beta_2 \rangle \varphi) & = & \mathbf{R}(Y \stackrel{\sigma}{=} \langle \beta_1 \rangle Y_1) \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} \langle \beta_2 \rangle \varphi) \\
\mathbf{R}(Y \stackrel{\sigma}{=} [\beta_1.\beta_2] \varphi) & = & \mathbf{R}(Y \stackrel{\sigma}{=} [\beta_1] Y_1) \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} [\beta_2] \varphi) \\
\mathbf{R}(Y \stackrel{\sigma}{=} \langle \beta_1 | \beta_2 \rangle \varphi) & = & \{Y \stackrel{\sigma}{=} Y_1 \vee Y_2\} \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} \langle \beta_1 \rangle \varphi) \cup \mathbf{R}(Y_2 \stackrel{\sigma}{=} \langle \beta_2 \rangle \varphi) \\
\mathbf{R}(Y \stackrel{\sigma}{=} [\beta_1 | \beta_2] \varphi) & = & \{Y \stackrel{\sigma}{=} Y_1 \wedge Y_2\} \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} [\beta_1] \varphi) \cup \mathbf{R}(Y_2 \stackrel{\sigma}{=} [\beta_2] \varphi) \\
\mathbf{R}(Y \stackrel{\sigma}{=} \langle \beta^* \rangle \varphi) & = & \{Y \stackrel{\sigma}{=} \varphi \vee Y_1\} \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} \langle \beta \rangle Y) \\
\mathbf{R}(Y \stackrel{\sigma}{=} [\beta^*] \varphi) & = & \{Y \stackrel{\sigma}{=} \varphi \wedge Y_1\} \cup \mathbf{R}(Y_1 \stackrel{\sigma}{=} [\beta] Y)
\end{array}
$$

Fig. 5. Translation of simple PDLR specifications in HMLR

For the simple PDLR specification obtained in the previous example, the translation **R** yields the following (simple) HMLR specification:

$$
\begin{array}{l}
\big(Y_0, \{Y_0 \stackrel{\nu}{=} Y_4 \wedge Y_5, Y_4 \stackrel{\nu}{=} [\mathtt{Send}]\, Y_1, Y_5 \stackrel{\nu}{=} [\mathsf{T}]\, Y_0\}. \\
\quad \{Y_1 \stackrel{\mu}{=} Y_2 \wedge Y_3, Y_2 \stackrel{\mu}{=} \langle \mathsf{T} \rangle\, \mathsf{T}, Y_3 \stackrel{\mu}{=} [\neg \mathtt{Recv}]\, Y_1\}\big)
\end{array}
$$

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 $\rho : \mathcal{Y} \to 2^S$. Moreover, it is easy to see that **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 **B** defined inductively in Figure 7. To every propositional variable Y in the left-hand side of an equation and to

Syntax of a BES:

$E = (x, B_1 \ldots\ldots B_p)$

where $B_j = \{x_{j\,i} \stackrel{\sigma_j}{=} op_{j\,i} X_{j\,i}\}_{1 \le i \le n_j}$, $x_{j\,i} \in \mathcal{X}$, $op_{j\,i} \in \{\vee, \wedge\}$, and $X_{j\,i} \subseteq \mathcal{X}$ for all $1 \le j \le p, 1 \le i \le n_j$

Semantics w.r.t. $\mathbf{Bool} = \{\mathsf{F}, \mathsf{T}\}$ and a context $\delta : \mathcal{X} \to \mathbf{Bool}$:

$$[\![(x, B_1 \ldots\ldots B_p)]\!]\, \delta = (\delta \oslash [\![B_1 \ldots\ldots B_p]\!]\, \delta)(x)$$

$$[\![B_j \ldots\ldots B_p]\!]\, \delta = ([\![B_j]\!]\, (\delta \oslash [\![B_{j+1} \ldots\ldots B_p]\!]\, \delta)) . [\![B_{j+1} \ldots\ldots B_p]\!]\, \delta$$

$$[\![\{x_{j\,i} \stackrel{\sigma_j}{=} op_{j\,i} X_{j\,i}\}_{1 \le i \le n_j}]\!]\, \delta = [\sigma_j \overline{\Psi}_{j\,\delta} / (x_{j\,1}, \ldots, x_{j\,n_j})]$$

where $[\![op\{x_1, ..., x_k\}]\!]\, \delta = \delta(x_1)\ op \ldots op\ \delta(x_k)$ and $\overline{\Psi}_{j\,\delta}$:$\mathbf{Bool}^{n_j} \to \mathbf{Bool}^{n_j}$, $\overline{\Psi}_{j\,\delta}(b_1, ..., b_{n_j}) = ([\![op_{j\,i} X_{j\,i}]\!](\delta \oslash [b_1/x_1, ..., b_{n\,j}/x_{n\,j}]))_{1 \le i \le n_j}$

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 HML formula φ in a right-hand side and to every state s is associated a boolean formula $\mathbf{B}(\varphi, s)$ encoding the fact that s satisfies φ.

$$\mathbf{B}(Y, M_1 \ldots\ldots M_p) = (Y_{s_0}, \mathbf{B}(M_1) \ldots\ldots \mathbf{B}(M_p))$$

$$\mathbf{B}(\{Y_i \stackrel{\sigma}{=} \varphi_i\}_{1 \le i \le n}) = \{Y_{i,s} \stackrel{\sigma}{=} \mathbf{B}(\varphi_i, s)\}_{1 \le i \le n, s \in S}$$

$$\begin{aligned}
\mathbf{B}(\mathsf{F}, s) &= \mathsf{F} \\
\mathbf{B}(\mathsf{T}, s) &= \mathsf{T} \\
\mathbf{B}(\varphi_1 \vee \varphi_2, s) &= \mathbf{B}(\varphi_1, s) \vee \mathbf{B}(\varphi_2, s) \\
\mathbf{B}(\varphi_1 \wedge \varphi_2, s) &= \mathbf{B}(\varphi_1, s) \wedge \mathbf{B}(\varphi_2, s) \\
\mathbf{B}(\langle \alpha \rangle\, \varphi, s) &= \textstyle\bigvee_{\{s \xrightarrow{a} s' \mid a \models \alpha\}} \mathbf{B}(\varphi, s') \\
\mathbf{B}([\alpha]\, \varphi, s) &= \textstyle\bigwedge_{\{s \xrightarrow{a} s' \mid a \models \alpha\}} \mathbf{B}(\varphi, s') \\
\mathbf{B}(Y_i, s) &= Y_{i,s}
\end{aligned}$$

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.

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 φ 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 σ-block of the BES.

For simplicity, we consider here the resolution of BESs containing a single μ-block (the solving routine for ν-blocks is completely dual). Multiple-block alternation-free BESs can be handled by associating to each σ-block in the BES its corresponding solving routine. Every time a variable x_j bound in a σ-block B_j is required in another block B_i 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; $L : V \to \{\vee, \wedge\}$ 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 $\{x_0, x_5, x_8\}$.

Every EBG $G = (V, E, L, F)$ induces a Kripke structure $\mathbf{G} = (V, E, L)$. 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 $\mathbf{B}$ given in Figure 7).

Let $P_\vee$ and $P_\wedge$ be two atomic propositions denoting the $\vee$- and $\wedge$-vertices of a Kripke structure $\mathbf{G}$ induced by a BES. The BES solution can be characterized by the following μ-calculus formula interpreted over $\mathbf{G}$ [24]:

$$\text{EX} = \mu Y.(P_\vee \wedge \langle \mathsf{T} \rangle\, Y) \vee (P_\wedge \wedge [\mathsf{T}]\, Y)$$

[1] This resolution scheme could be naturally implemented using coroutines.

$$\begin{cases} x_0 \stackrel{\mu}{=} x_1 \vee x_4 \vee x_9 \\ x_1 \stackrel{\mu}{=} x_2 \wedge x_3 \\ x_2 \stackrel{\mu}{=} x_1 \wedge x_3 \\ x_3 \stackrel{\mu}{=} \mathsf{T} \\ x_4 \stackrel{\mu}{=} x_3 \wedge x_5 \\ x_5 \stackrel{\mu}{=} x_3 \vee x_6 \\ x_6 \stackrel{\mu}{=} x_4 \wedge x_6 \\ x_7 \stackrel{\mu}{=} \mathsf{F} \\ x_8 \stackrel{\mu}{=} x_0 \vee x_7 \\ x_9 \stackrel{\mu}{=} x_7 \wedge x_8 \end{cases}$$

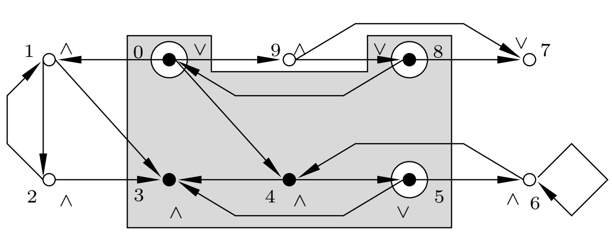

Fig. 8. A BES, its associated EBG, and a subgraph

A variable x of the BES is true iff the vertex x satisfies EX in $\mathbf{G}$, noted $x \models_{\mathbf{G}}$ EX. Intuitively, 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 $P_\wedge \wedge [\mathsf{T}]\,\mathsf{F}$). 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 EX on $\mathbf{G}$, noted $[\![\text{EX}]\!]_{\mathbf{G}}$. Thus, the local resolution of a BES amounts to the local model-checking of the EX formula on the corresponding Kripke structure.

Consider an EBG $G = (V, E, L, \emptyset)$, its associated Kripke structure $\mathbf{G} = (V, E, L)$, and $x \in V$. The local model-checking of EX on x does not always require to entirely explore $\mathbf{G}$ (e.g., on Figure 8, one could explore only the outlined subgraph in order to check EX on x_0), but rather to explore a part $\mathbf{G}'$ of $\mathbf{G}$ such that the value of x can be computed based only on the information in $\mathbf{G}'$. 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 EX by x is the same in $\mathbf{G}'$ and $\mathbf{G}$: $[\![\text{EX}]\!]_{\mathbf{G}'} = [\![\text{EX}]\!]_{\mathbf{G}} \cap V'$. A subgraph G' is solution-closed iff the satisfaction of EX on its frontier F' can be decided using only the information in G': $F' \subseteq [\![(P_\vee \wedge \text{EX}) \vee (P_\wedge \wedge \neg\text{EX})]\!]_{\mathbf{G}'}$. For the EBG on Figure 8, it is easy to see that the subgraph outlined is solution-closed: its frontier $\{x_0, x_5, x_8\}$ contains only $\vee$-vertices satisfying EX.

Local resolution algorithm. The SOLVE algorithm that we propose (see Figure 9) takes as input an implicit Kripke structure $\mathbf{G} = (V, E, L)$ induced by an EBG G and a vertex $x \in V$ on which the EX formula must be checked. Starting from x, SOLVE performs a depth-first search (DFS) of $\mathbf{G}$ and simultaneously checks EX on all visited vertices, which are stored in a set $A \subseteq V$. Upon termination, the subgraph G_A of G containing all vertices in A and all edges traversed during the DFS is solution-closed ($[\![\text{EX}]\!]_{\mathbf{G_A}} = [\![\text{EX}]\!]_{\mathbf{G}} \cap A$), meaning that the truth value of 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 SOLVE (x, (V, E, L)) is
    var A, B : 2^V; d : V → 2^V; c, p : V → Nat;
          y, z, u, w : V; stack : V*;
    c(x) := if L(x) = ∧ then |E(x)| else 1;
    p(x) := 0; d(x) := ∅;
    A := {x}; stack := push(x, nil);
    while stack ≠ nil do
        y := top(stack);
        if c(y) = 0 then
            if d(y) ≠ ∅ then
                B := {y};
                while B ≠ ∅ do
                    let u ∈ B; B := B \ {u};
                    forall w ∈ d(u) do
                        if c(w) > 0 then
                            c(w) := c(w) − 1;
                            if c(w) = 0 then
                                B := B ∪ {w}
                            endif
                        endif
                    end;
                    d(u) := ∅
                end
            else
                stack := pop(stack)
            endif
        elsif p(y) ≤ |E(y)| − 1 then
            z := (E(y))_p(y); p(y) := p(y) + 1;
            if z ∈ A then
                d(z) := d(z) ∪ {y}
                if c(z) = 0 then
                    stack := push(z, stack)
                endif
            else
                c(z) := if L(z) = ∧ then |E(z)| else 1
                p(z) := 0; d(z) := {y};
                A := A ∪ {z}; stack := push(z, stack)
            endif
        else
                stack := pop(stack)
        endif
    end
end
```

Fig. 9. Graph-based local resolution of a BES with sign μ

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 $y \models_{\mathbf{G}}$ Ex is encountered on top of the stack (this can be either a "new" $\wedge$-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 $u \models_{\mathbf{G_A}}$ 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 $y \models_{\mathbf{G_A}}$ Ex ($c(y)$ is initialized to 1 for $\vee$-vertices and to $|E(y)|$ for $\wedge$-vertices). Thus, for every $y \in A$, $y \models_{\mathbf{G_A}}$ Ex iff $c(y) = 0$.

Figure 10 shows the result of executing Solve for the variable x_0 and the Ebg 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 $\vee$-vertices satisfying Ex in $\mathbf{G_A}$.

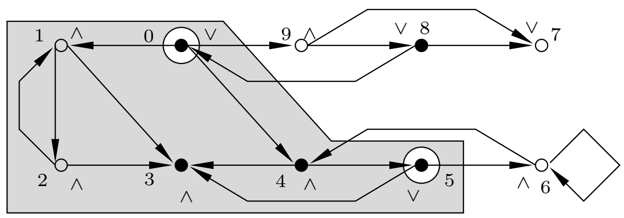

Fig. 10. A solution-closed subgraph computed by Solve

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 $\mathbf{G}$ 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

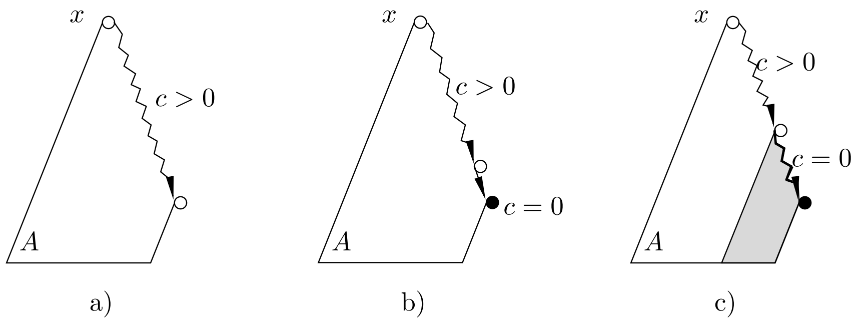

Fig. 11. Structure of the DFS stack during the execution of SOLVE

become stable: therefore, 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 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).

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 DFS) and backwards (when EX is reevaluated on its source vertex). Moreover, SOLVE has also a good average-case complexity, improving on Andersen and Vergauwen-Lewi's algorithms, since it stops as soon as $x \models_{\mathbf{G_A}}$ 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 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 EBG 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 EX by x. A diagnostic G' is called *example* if $x \models_{\mathbf{G'}}$ EX and *counterexample* if $x \not\models_{\mathbf{G'}}$ EX.

[2] The reevaluation of EX, which involves a backwards traversal of edges in G_A, can affect only those vertices in the DFS tree that are descendants of stable vertices present on the stack, outlined by the grey portion on Figure 11 c).

The 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 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 SOLVE.

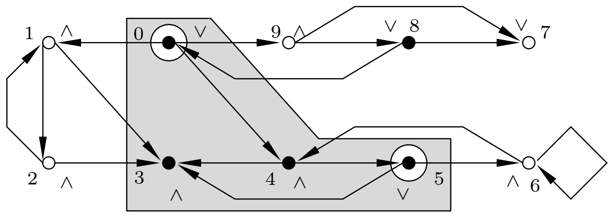

Fig. 12. An example for x_0

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 SOLVE.

4 Implementation and use

We used the model-checking method presented in Section 3 as a basis for developing the EVALUATOR 3.0 model-checker within the CADP (CÆSAR/ALDÉBARAN) toolset [9]. The tool has been built using the OPEN/CÆSAR environment [13], which provides a generic API for on-the-fly exploration of transition systems. As a consequence, EVALUATOR 3.0 can be used in conjunction with every compiler that is OPEN/CÆSAR–compliant (i.e., that implements a translation from its input language to the OPEN/CÆSAR API), and particularly with the CÆSAR compiler [14] for 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 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 μ-calculus. For example, the $\mathsf{E}\left[\varphi_{1\,\alpha_1}\mathsf{U}_{\alpha_2}\varphi_2\right]$ operator of ACTL (stating the existence of a sequence $s_1 \stackrel{a_1}{\to} s_2 \stackrel{a_2}{\to} \cdots s_k \stackrel{a_k}{\to} s_{k+1}$ such that $s_i \models \varphi_1$ for all $1 \le i \le k$, $a_j \models \alpha_1$ for all $1 \le j < k$, $a_k \models \alpha_2$, and $s_{k+1} \models \varphi_2$) can be encoded as a macro $\mathsf{EU\_A\_A}(\varphi_1, \alpha_1, \alpha_2, \varphi_2) = \mu Y.(\varphi_1 \wedge (\langle\alpha_2\rangle\,\varphi_2 \vee \langle\alpha_1\rangle\,Y))$. 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/CÆSAR 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 μ-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 μ-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 LTSs, by checking PDL basic modalities: a transition sequence starting at the initial state and satisfying a regular formula β can be obtained either as an example for the $\langle\beta\rangle\,\mathsf{T}$ formula, or as a counterexample for the $[\beta]\,\mathsf{F}$ formula.

Table 2. Property patterns in regular alternation-free μ-calculus

PATTERN	SCOPE	FORMULA
Absence (α_1 is false)	Globally	$[\mathsf{T}^*.\alpha_1]\,\mathsf{F}$
	Before α_2	$[(\neg\alpha_2)^*.\alpha_1.\mathsf{T}^*.\alpha_2]\,\mathsf{F}$
	After α_2	$[(\neg\alpha_2)^*.\alpha_2.\mathsf{T}^*.\alpha_1]\,\mathsf{F}$
	Between α_2 and α_3	$[\mathsf{T}^*.\alpha_2.(\neg\alpha_3)^*.\alpha_1.\mathsf{T}^*.\alpha_3]\,\mathsf{F}$
	After α_2 until α_3	$[\mathsf{T}^*.\alpha_2.(\neg\alpha_3)^*.\alpha_1]\,\mathsf{F}$
Existence (α_1 becomes true)	Globally	$\mu Y.\,\langle\mathsf{T}\rangle\,\mathsf{T} \wedge [\neg\alpha_1]\,Y$
	Before α_2	$[(\neg\alpha_1)^*.\alpha_2]\,\mathsf{F}$
	After α_2	$[(\neg\alpha_2)^*.\alpha_2]\,\mu Y.\,\langle\mathsf{T}\rangle\,\mathsf{T} \wedge [\neg\alpha_1]\,Y$
	Between α_2 and α_3	$[\mathsf{T}^*.\alpha_2.(\neg\alpha_1)^*.\alpha_3]\,\mathsf{F}$
	After α_2 until α_3	$[\mathsf{T}^*.\alpha_2]\,([(\neg\alpha_1)^*.\alpha_3]\,\mathsf{F} \wedge \mu Y.\,\langle\mathsf{T}\rangle\,\mathsf{T} \wedge [\neg\alpha_1]\,Y)$
Universality (α_1 is true)	Globally	$[\mathsf{T}^*.\neg\alpha_1]\,\mathsf{F}$
	Before α_2	$[(\neg\alpha_2)^*.\neg(\alpha_1 \vee \alpha_2).(\neg\alpha_2)^*.\alpha_2]\,\mathsf{F}$
	After α_2	$[(\neg\alpha_2)^*.\alpha_2.\mathsf{T}^*.\neg\alpha_1]\,\mathsf{F}$
	Between α_2 and α_3	$[\mathsf{T}^*.\alpha_2.(\neg\alpha_3)^*.\neg(\alpha_1 \vee \alpha_3).\mathsf{T}^*.\alpha_3]\,\mathsf{F}$
	After α_2 until α_3	$[\mathsf{T}^*.\alpha_2.(\neg\alpha_3)^*.\neg(\alpha_1 \vee \alpha_3)]\,\mathsf{F}$

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 $\texttt{Put}_i$ and $\texttt{Get}_i$ denote the communication of message i on gates `Put` and `Get`, respectively. Predicates $\texttt{Put}_{any}$ and $\texttt{Get}_{any}$ (wildcards) denote the communication of arbitrary messages on gates `Put` and `Get`. Every property containing an occurrence of $\texttt{Put}_i$ and/or $\texttt{Get}_i$ has been checked for all values of i between 0 and n.

Table 3. Properties of the Alternating Bit Protocol

No.	Property	Formula	Value
P_1	Initially, a `Put` will be eventually reached	$\mu Y. \langle \mathsf{T} \rangle\, \mathsf{T} \wedge [\neg \texttt{Put}_{any}]\, Y$	false
P_2	Initially, a `Put` will be fairly reached	$[(\neg \texttt{Put}_{any})^*]\, \langle \mathsf{T}^*.\texttt{Put}_{any} \rangle\, \mathsf{T}$	true
P_3	Initially, no `Get` can be reached before the corresponding `Put`	$[(\neg \texttt{Put}_i)^*.\texttt{Get}_i]\, \mathsf{F}$	true
P_4	Between two consecutive `Put`, there is a corresponding `Get`	$[\mathsf{T}^*.\texttt{Put}_i.(\neg \texttt{Get}_i)^*.\texttt{Put}_{any}]\, \mathsf{F}$	true
P_5	Between two consecutive `Get`, there is a corresponding `Put`	$[\mathsf{T}^*.\texttt{Get}_{any}.(\neg \texttt{Put}_i)^*.\texttt{Get}_i]\, \mathsf{F}$	true
P_6	After a `Put`, the corresponding `Get` is eventually reachable	$[\mathsf{T}^*.\texttt{Put}_i]\, \mu Y. \langle \mathsf{T} \rangle\, \mathsf{T} \wedge [\neg \texttt{Get}_i]\, Y$	false
P_7	After a `Put`, the corresponding `Get` is fairly reachable	$[\mathsf{T}^*.\texttt{Put}_i.(\neg \texttt{Get}_i)^*]\, \langle (\neg \texttt{Get}_i)^*.\texttt{Get}_i \rangle\, \mathsf{T}$	true

Properties P_1 and P_6, which express the inevitable reachability of `Put` and `Get` actions, are false because of the livelocks (τ-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 μ-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_6, 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

No.		$n = 20$ $\lvert S\rvert = 39\,800$		$n = 40$ $\lvert S\rvert = 153\,200$		$n = 60$ $\lvert S\rvert = 340\,200$		$n = 80$ $\lvert S\rvert = 600\,800$		$n = 100$ $\lvert S\rvert = 935\,000$	
		time	*expl.%*	*time*	*expl.%*	*time*	*expl.%*	*time*	*expl.%*	*time*	*expl.%*
$\boldsymbol{P_1}$	*a*	**0″**	**0.01**	**0″**	**0.00**	**0″**	**0.00**	**0″**	**0.00**	**0″**	**0.00**
	b	**20″**	**93.1**	**1′42″**	**96.4**	**4′49″**	**97.6**	**10′04″**	**98.2**	**18′23″**	**98.5**
$\boldsymbol{P_2}$	*a*	**0″**	**0.01**	**0″**	**0.00**	**0″**	**0.00**	**0″**	**0.00**	**0″**	**0.00**
	b	**1′02″**	**100**	**5′11″**	**100**	**14′29″**	**100**	**30′59″**	**100**	**56′28″**	**100**
P_3	*a*	8″	91.7	35″	95.7	1′20″	97.1	2′28″	97.8	4′03″	98.2
	b	16″	91.7	1′09″	95.7	2′53″	97.1	5′49″	97.8	9′57″	98.2
P_4	*a*	9″	100	37″	100	1′25″	100	2′35″	100	4′13″	100
	b	19″	100	1′14″	100	3′05″	100	6′05″	100	10′17″	100
P_5	*a*	18″	100	1′15″	100	2′58″	100	5′48″	100	10′07″	100
	b	38″	100	3′01″	100	8′20″	100	17′40″	100	31′53″	100
$\boldsymbol{P_6}$	*a*	**0″**	**0.02**	**0″**	**0.00**	**0″**	**0.00**	**0″**	**0.00**	**0″**	**0.00**
	b	**48″**	**100**	**3′34″**	**100**	**9′16″**	**100**	**18′54″**	**100**	**33′26″**	**100**
P_7	*a*	10″	100	38″	100	1′26″	100	2′36″	100	4′15″	100
	b	20″	100	1′18″	100	3′06″	100	6′08″	100	10′23″	100

(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 μ-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/ALDÉBARAN) 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 μ-calculus. At the present time, we developed libraries encoding the operators of CTL [4], 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/preorder 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 μCRL [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

1. H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, April 1994.
2. A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29:57–66, 1988.
3. H. Bekić. *Definable operations in general algebras, and the theory of automata and flowcharts.* volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer Verlag, Berlin, 1984.
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
5. R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. In K. G. Larsen and A. Skou, editors, *Proceedings of 3rd Workshop on Computer Aided Verification CAV '91 (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58, Berlin, July 1991. Springer Verlag.
6. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering ICSE'99 (Los Angeles, CA, USA)*, May 1999.
7. E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of the 1st LICS*, pages 267–278, 1986.
8. A. Fantechi, S. Gnesi, F. Mazzanti, R. Pugliese, and E. Tronci. A Symbolic Model Checker for ACTL. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods FM-Trends'98 (Boppard, Germany)*, volume 1641 of *Lecture Notes in Computer Science*. Springer Verlag, October 1998.
9. Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.
10. Jean-Claude Fernandez, Claude Jard, Thierry Jéron, Laurence Nedelka, and César Viho. Using On-the-Fly Verification Techniques for the Generation of Test Suites. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (Rutgers University, New Brunswick, NJ, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer Verlag, August 1996. Also available as INRIA Research Report RR-2987.
11. Jean-Claude Fernandez and Laurent Mounier. A Local Checking Algorithm for Boolean Equation Systems. Rapport SPECTRE 95-07, VERIMAG, Grenoble, March 1995.
12. M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, (18):194–211, 1979.
13. Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.
14. Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International*

Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada), pages 379–394. IFIP, North-Holland, June 1990.

15. J-F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, December 1990.
16. A. Ingolfsdottir and B. Steffen. Characteristic Formulae for Processes with Divergence. *Information and Computation*, 110(1):149–163, June 1994.
17. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.
18. D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
19. K. G. Larsen. Proof Systems for Hennessy-Milner logic with Recursion. In *Proceedings of the 13th Colloquium on Trees in Algebra and Programming CAAP '88 (Nancy, France)*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230, Berlin, March 1988. Springer Verlag.
20. X. Liu, C. R. Ramakrishnan, and S. A. Smolka. Fully Local and Efficient Evaluation of Alternating Fixed Points. In Bernhard Steffen, editor, *Proceedings of 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *LNCS*, pages 5–19, Berlin, March 1998. Springer Verlag.
21. Angelika Mader. *Verification of Modal Properties Using Boolean Equation Systems.* VERSAL 8, Bertz Verlag, Berlin, 1997.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume I (Specification). Springer Verlag, 1992.
23. R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In Annalisa Bossi, Agostino Cortesi, and Francesca Levi, editors, *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation VMCAI'98 (Pisa, Italy)*. University Ca' Foscari of Venice, September 1998.
24. Radu Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In Susanne Graf and Michael Schwartzbach, editors, *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000 (Berlin, Germany)*, volume 1785 of *Lecture Notes in Computer Science*, pages 251–265. Springer Verlag, March 2000. Full version available as INRIA Research Report RR-3861.
25. R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, 1990.
26. D. Simon, B. Espiau, K. Kapellos, R. Pissard-Gibollet, and al. The Orccad Architecture. *International Journal of Robotics Research*, 17(4):338–359, April 1998.
27. R. Streett. Propositional Dynamic Logic of Looping and Converse. *Information and Control*, (54):121–141, 1982.
28. E. Tronci. Hardware Verification, Boolean Logic Programming, Boolean Functional Programming. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science LICS'95 (San Diego, California)*, pages 408–418. IEEE Computer Society Press, June 1995.
29. B. Vergauwen and J. Lewi. Efficient Local Correctness Checking for Single and Alternating Boolean Equation Systems. In S. Abiteboul and E. Shamir, editors, *Proceedings of the 21st ICALP (Vienna)*, volume 820 of *Lecture Notes in Computer Science*, pages 304–315, Berlin, July 1994. Springer Verlag.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399