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

Radu Mateescu [a,1], Mihaela Sighireanu [b,2]

[a] *INRIA Rhône-Alpes / VASY, 655, avenue de l'Europe, F-38330 Montbonnot Saint Martin, France*

[b] *Université Paris 7 / LIAFA, 2, place Jussieu, F-75251 Paris, France*

Abstract

Model-checking is a successful technique for automatically verifying concurrent finite-state systems. When designing 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 toolbox using the generic OPEN/CAESAR environment for on-the-fly verification.

Key words: Boolean equation system, Diagnostic, Labeled transition system, Model-checking, Mu-calculus, Specification, Temporal logic, Verification

1 Introduction

Formal verification is essential in order to improve the reliability of complex, critical applications such as communication protocols and distributed systems.

[1] E-mail: Radu.Mateescu@inria.fr

[2] E-mail: Mihaela.Sighireanu@liafa.jussieu.fr

Article published in Science of Computer Programming 46 (2003) 255–281

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 [32] is particularly powerful, subsuming linear-time logics such as LTL [37], branching-time logics such as CTL [8] or ACTL [13], and regular logics such as PDL [22] or PDL-Δ [43].

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 studied. Among these, the *alternation-free* fragment [17] makes a good compromise between expressiveness (it allows e.g., direct encodings of CTL and ACTL) and efficiency of the verification (there are several model-checking algorithms with linear-time complexity [10,2,47,34]).

Thirdly, the model-checker interface should allow an intuitive, concise, and flexible description of properties, in order to reduce the risk of specification errors and to facilitate the verification task for non-expert users. Moreover, the model-checker must offer enough feedback information to enable the debugging of applications; 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 linear-time local algorithm based on a depth-first search of the corresponding boolean graph. Compared to classical linear-time local algorithms [2,47], 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 [39], 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 toolbox [19] using the generic OPEN/CAESAR environment for on-the-fly verification [23].

The extension of temporal logics with regular operators has been extensively studied in the literature. As regards linear-time logics, the first extension of LTL with operators defined by means of regular grammars was proposed in [48], leading to a strictly more expressive logic called ETL. More elaborate extensions of LTL with various types of automata were studied in [49,46]. As regards branching-time logics, extensions of CTL and CTL* with Büchi automata have been proposed in [27] and [44], respectively. Since we aim to be adequate with action-based description formalisms like process algebras and related languages like LOTOS [31], in this paper we focused on branching-time, action-based logics such as the modal μ-calculus. The idea of extending the alternation-free μ-calculus with the regular modalities of PDL has been put forward in [6]. Although theoretically this extension does not increase the expressive power of the alternation-free μ-calculus (since this logic can encode PDL modalities [17]), in practice it significantly improves the readability of formulas, by allowing in many cases to replace complex fixed point formulas by regular modalities.

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. Section 4 discusses the implementation of the model-checker within the CADP toolbox and presents several applications. 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 [32,17] with action formulas as in ACTL [13] and with regular expressions over action sequences as in PDL [22]. It allows direct encodings of "pure" branching-time logics like ACTL or CTL [8], as well as of regular logics like PDL or PDL-Δ [43]. 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 labeled transition systems (LTSs), which are 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 evolve from state s to state s' by performing action a.

The regular alternation-free μ-calculus contains three types of formulas, namely *action formulas* (noted α), *regular formulas* (noted β), and *state formulas* (noted φ), as expressed by the grammar in Table 1. Action formulas are built upon action names $a \in A$ and the standard boolean operators. Derived boolean connectives are defined as usual: $\mathrm{F} = a \wedge \neg a$ for some a, $\mathrm{T} = \neg\mathrm{F}$, $\alpha_1 \vee \alpha_2 = \neg(\neg\alpha_1 \wedge \neg\alpha_2)$, etc. Regular formulas are built upon action formulas and the standard regular expression operators, namely concatenation (.), choice (|), and transitive-reflexive closure ($*$). The empty sequence operator ε and the transitive closure operator $+$ are defined as $\varepsilon = \mathrm{F}^*$ and $\beta^+ = \beta.\beta^*$. State formulas are built upon propositional variables $Y \in \mathcal{Y}$ and the standard boolean operators, the possibility and necessity modal 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 said *closed*.

Table 1
Syntax of regular alternation-free μ-calculus

$\alpha ::= a \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2$
$\beta ::= \alpha \mid \beta_1.\beta_2 \mid \beta_1\|\beta_2 \mid \beta^*$
$\varphi ::= \mathrm{F} \mid \mathrm{T} \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle\beta\rangle\,\varphi \mid [\beta]\,\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi$

State formulas are assumed to be *alternation-free* [17], which intuitively means that mutual recursion between minimal and maximal fixed point variables is forbidden. For our logic, this syntactic condition is more subtle than for the standard alternation-free μ-calculus because of the $\langle\beta\rangle\,\varphi$ and $[\beta]\,\varphi$ modalities with $*$ operators inside β, which are equivalent to "hidden" minimal and maximal fixed point formulas, respectively. A state formula is alternation-free if for every fixed point subformula $\mu Y.\varphi$, the variable Y has no free occurrences inside a subformula of φ of the form $\nu Z.\varphi'$ or $[\beta]\,\varphi'$ where β contains a $*$ operator (a dual condition holds for maximal fixed point subformulas $\nu Y.\varphi$).

The semantics of the logic is shown in Table 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

Table 2
Semantics of regular alternation-free μ-calculus

$$\begin{array}{rlrl}
[\![a]\!] &= \{a\} & \|\alpha\| &= \{(s,s') \in S \times S \mid \exists a \in [\![\alpha]\!] \,.\, s \xrightarrow{a} s'\} \\
[\![\neg\alpha]\!] &= A \setminus [\![\alpha]\!] & \|\beta_1.\beta_2\| &= \|\beta_1\| \circ \|\beta_2\| \\
[\![\alpha_1 \wedge \alpha_2]\!] &= [\![\alpha_1]\!] \cap [\![\alpha_2]\!] & \|\beta_1|\beta_2\| &= \|\beta_1\| \cup \|\beta_2\| \\
& & \|\beta^*\| &= \|\beta\|^*
\end{array}$$

$$\begin{array}{rl}
[\![\mathrm{F}]\!]\,\rho &= \emptyset \\
[\![\mathrm{T}]\!]\,\rho &= S \\
[\![\varphi_1 \vee \varphi_2]\!]\,\rho &= [\![\varphi_1]\!]\,\rho \cup [\![\varphi_2]\!]\,\rho \\
[\![\varphi_1 \wedge \varphi_2]\!]\,\rho &= [\![\varphi_1]\!]\,\rho \cap [\![\varphi_2]\!]\,\rho \\
[\![\langle\beta\rangle\,\varphi]\!]\,\rho &= \{s \in S \mid \exists s' \in S.(s,s') \in \|\beta\| \wedge s' \in [\![\varphi]\!]\,\rho\} \\
[\![[\beta]\,\varphi]\!]\,\rho &= \{s \in S \mid \forall s' \in S.(s,s') \in \|\beta\| \Rightarrow s' \in [\![\varphi]\!]\,\rho\} \\
[\![Y]\!]\,\rho &= \rho(Y) \\
[\![\mu Y.\varphi]\!]\,\rho &= \bigcap\{S' \subseteq S \mid \Phi_\rho(S') \subseteq S'\} \\
[\![\nu Y.\varphi]\!]\,\rho &= \bigcup\{S' \subseteq S \mid S' \subseteq \Phi_\rho(S')\} \\
& \text{where } \Phi_\rho : 2^S \to 2^S,\ \Phi_\rho(S') = [\![\varphi]\!]\,(\rho \oslash [S'/Y])
\end{array}$$

target states of transition sequences satisfying β ($\circ$, $\cup$, and $*$ denote respectively 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 (the $\oslash$ notation denotes context overriding: $(\rho_1 \oslash \rho_2)(Y)$ is equal to $\rho_2(Y)$ if Y is assigned by ρ_2 and to $\rho_1(Y)$ otherwise). 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 $a \models \alpha$) 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 $[\![\varphi]\!] = S$. Because an on-the-fly model-checker only decides whether $s_0 \models \varphi$, the reader should be aware that verifying $L \models \varphi$ amounts to check on-the-fly the formula $[\mathrm{T}^*]\,\varphi$ (which is equivalent to the ACTL formula $\mathbf{AG}_\mathrm{T}\varphi$ or the CTL formula $\mathbf{AG}\varphi$), stating that φ holds on every state reachable from s_0.

The regular alternation-free μ-calculus allows to express intuitively and concisely various useful properties of LTSs. Table 3 shows some typical examples of formulas. Usual safety properties, stating the absence of "bad" execution sequences characterized by regular formulas β, can be encoded by a single box modality $[\beta]\,\mathrm{F}$. Basic liveness properties, stating the existence of "good" execution sequences characterized by β, can be encoded by a single diamond modality $\langle\beta\rangle\,\mathrm{T}$; more complex properties, stating the existence of certain branching patterns (e.g., inevitable reachability) can be expressed using minimal fixed point operators. Some fairness properties, such as reachability over strongly fair execution sequences (i.e., by exiting circuits after a finite number of steps), can be encoded using nested box and diamond modalities.

Note that boolean connectives (particularly $\neg$) over actions improve the conciseness of the formulas, allowing for instance to express the inevitable reachability of an action without referring to other actions in the LTS. Also, regular operators improve readability: without these operators, the second liveness property given in Table 3 would be described by the equivalent formula $\mu Y_1.(\langle\texttt{Send}\rangle\,\mu Y_2.(\langle\texttt{Recv}\rangle\,\mathrm{T} \vee \mu Y_3.(\langle\texttt{Error}\rangle\,Y_2 \vee \langle\mathrm{T}\rangle\,Y_3)) \vee \langle\mathrm{T}\rangle\,Y_1)$.

Table 3
Examples of properties in regular alternation-free μ-calculus

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

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 variables bound by fixed point operators are distinct). The verification problem we are interested in consists of deciding whether $s_0 \models \varphi$. An efficient method used for the ACTL logic [18] and for the alternation-free μ-calculus [10,2] is to translate the problem into a boolean equation system (BES) [2,36], which is solved using specific local algorithms [2,47,45]. 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 would mean 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 [17]. This translation is succinct (it produces at most a linear blow-up in the size of φ), but rather tedious to apply in practice because it requires the identification and sharing of common subformulas. On the other hand, since we seek to reduce the verification problem $s_0 \models \varphi$ to a BES resolution, it seems more natural to use an equation-based intermediate representation instead of a formula-based one as in [17]. Moreover, this allows to devise a simpler, yet succinct translation of the verification problem into a BES resolution without identifying common subformulas.

The translation that we propose consists of three steps: (a) translation of a formula φ into a fixed point equation system containing PDL modalities; (b) simplification of the equation system by translating PDL modalities into HML modalities; and (c) translation of the model-checking problem of the resulting system over an LTS into a BES resolution. The first two steps are purely syntactic, i.e., they take into account only the temporal logic specification, whereas the third one involves semantic information contained in the LTS. The following three sections describe in detail each translation step.

3.1.1 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 [33]. A PDLR specification (see Table 4)

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 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 $1 \leq j < p$) 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 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.

Table 4
Syntax and semantics of PDLR

Syntax of a PDLR specification:

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

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

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

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

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

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

where $\overline{\Phi}_{j_\rho} : (2^S)^{n_j} \to (2^S)^{n_j}$,

$$\overline{\Phi}_{j_\rho}(U_1, \ldots, U_{n_j}) = ([\![\varphi_i]\!]\,(\rho \oslash [U_1/Y_1, \ldots, U_{n_j}/Y_{n_j}]))_{1 \leq i \leq n_j}$$

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 $\mathcal{Y} \to 2^S$ obtained by concatenating the solutions of all σ-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 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 equivalent to "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 $(\mathbf{T}_\mathbf{1}^\sigma(\sigma Y_0 . \varphi_0), \mathbf{T}_\mathbf{2}^\sigma(\sigma Y_0 . \varphi_0))$ is obtained using two syntactic functions $\mathbf{T_1}$ and

$\mathbf{T_2}$, defined inductively in Table 5. $\mathbf{T}_\mathbf{1}^\sigma\varphi$ yields a formula obtained from φ by substituting each fixed point subformula by its corresponding variable. $\mathbf{T}_\mathbf{2}^\sigma\varphi$ 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 $hd(\mathbf{T}_\mathbf{2}^\sigma\varphi)$, contains the equations of sign σ associated to the topmost fixed point subformulas of φ. The remainder of the system, denoted by $tl(\mathbf{T}_\mathbf{2}^\sigma\varphi)$, 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 ($\tilde{\mu} = \nu$, $\tilde{\nu} = \mu$).

Table 5
Translation of state formulas in PDLR

φ	$\mathbf{T}_\mathbf{1}^\sigma\varphi$	$\mathbf{T}_\mathbf{2}^\sigma\varphi$
F	F	$\{\,\}$
T	T	$\{\,\}$
$\langle\beta\rangle\,\varphi_1$	$\langle\beta\rangle\,\mathbf{T}_\mathbf{1}^\sigma\varphi_1$	$\mathbf{T}_\mathbf{2}^\sigma\varphi_1$
$[\beta]\,\varphi_1$	$[\beta]\,\mathbf{T}_\mathbf{1}^\sigma\varphi_1$	$\mathbf{T}_\mathbf{2}^\sigma\varphi_1$
$\varphi_1 \vee \varphi_2$	$\mathbf{T}_\mathbf{1}^\sigma\varphi_1 \vee \mathbf{T}_\mathbf{1}^\sigma\varphi_2$	$(hd(\mathbf{T}_\mathbf{2}^\sigma\varphi_1) \cup hd(\mathbf{T}_\mathbf{2}^\sigma\varphi_2))\mathbf{.}tl(\mathbf{T}_\mathbf{2}^\sigma\varphi_1)\mathbf{.}tl(\mathbf{T}_\mathbf{2}^\sigma\varphi_2)$
$\varphi_1 \wedge \varphi_2$	$\mathbf{T}_\mathbf{1}^\sigma\varphi_1 \wedge \mathbf{T}_\mathbf{1}^\sigma\varphi_2$	$(hd(\mathbf{T}_\mathbf{2}^\sigma\varphi_1) \cup hd(\mathbf{T}_\mathbf{2}^\sigma\varphi_2))\mathbf{.}tl(\mathbf{T}_\mathbf{2}^\sigma\varphi_1)\mathbf{.}tl(\mathbf{T}_\mathbf{2}^\sigma\varphi_2)$
Y	Y	$\{\,\}$
$\sigma Y.\varphi_1$	Y	$(\{Y \stackrel{\sigma}{=} \mathbf{T}_\mathbf{1}^\sigma\varphi_1\} \cup hd(\mathbf{T}_\mathbf{2}^\sigma\varphi_1))\mathbf{.}tl(\mathbf{T}_\mathbf{2}^\sigma\varphi_1)$
$\tilde{\sigma}Y.\varphi_1$	Y	$\{\,\}\mathbf{.}(\{Y \stackrel{\tilde{\sigma}}{=} \mathbf{T}_\mathbf{1}^{\tilde{\sigma}}\varphi_1\} \cup hd(\mathbf{T}_\mathbf{2}^{\tilde{\sigma}}\varphi_1))\mathbf{.}tl(\mathbf{T}_\mathbf{2}^{\tilde{\sigma}}\varphi_1)$

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

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

The translation $(\mathbf{T}_\mathbf{1}^\nu\varphi, \mathbf{T}_\mathbf{2}^\nu\varphi)$ yields the PDLR specification below:

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

The functions $\mathbf{T_1}$ and $\mathbf{T_2}$ are similar to those proposed in [36, chap. 3] for translating nested fixed point expressions into equation systems. Using Bekić's theorem [5], it can be shown that the translation defined by $\mathbf{T_1}$ and $\mathbf{T_2}$ preserves the semantics of state formulas: $[\![\sigma Y.\varphi]\!]\,\rho = [\![(\mathbf{T}_\mathbf{1}^\sigma(\sigma Y.\varphi), \mathbf{T}_\mathbf{2}^\sigma(\sigma Y.\varphi))]\!]\,\rho$ 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:

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

3.1.2 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 (overloaded) syntactic functions $\mathbf{R}$ defined in Table 6. 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.

Table 6
Translation of simple PDLR specifications in HMLR

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

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

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

Intuitively, the function $\mathbf{R}$ applies (by taking care to keep a single operator in the right-hand sides of the equations) the well-known equivalences on PDL formulas [22,17]: $\langle \beta_1.\beta_2 \rangle \varphi = \langle \beta_1 \rangle \langle \beta_2 \rangle \varphi$, $\langle \beta_1 | \beta_2 \rangle \varphi = \langle \beta_1 \rangle \varphi \vee \langle \beta_2 \rangle \varphi$, $\langle \beta^* \rangle \varphi = \varphi \vee \langle \beta \rangle \langle \beta^* \rangle \varphi$ (and their dual counterparts for box modalities). Therefore, the translation $\mathbf{R}$ 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, $\mathbf{R}$ may cause at most a linear blow-up in the size of the equation system.

3.1.3 Translation of the model-checking problem into BESs

The third step is to translate the verification problem of a simple HMLR specification on an LTS into the local resolution of an alternation-free boolean equation system (BES). A BES (see Table 7) 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, empty disjunctions and conjunctions being equivalent to F and T, respectively. One can easily transform a BES with arbitrary formulas into this simple form by introducing new variables and equations associated to nested subformulas, at the price of at most a linear blow-up in the size of the system [3,2,47]. 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 $B_1 . \ldots . B_p$.

Table 7
Syntax and semantics of boolean equation systems

Syntax of a BES:

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

where $B_j = \{x_{j_i} \stackrel{\sigma_j}{=} op_{j_i} X_{j_i}\}_{1 \leq i \leq n_j}$, $x_{j_i} \in \mathcal{X}$, $op_{j_i} \in \{\vee, \wedge\}$, and $X_{j_i} \subseteq \mathcal{X}$

for all $1 \leq j \leq p, 1 \leq i \leq n_j$

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

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

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

$$[\![\{x_{j_i} \stackrel{\sigma_j}{=} op_{j_i} X_{j_i}\}_{1 \leq i \leq n_j}]\!] \delta = [\sigma_j \overline{\Psi}_{j\delta} / (x_{j_1}, \ldots, x_{j_{n_j}})]$$

where $[\![op\{x_1, ..., x_k\}]\!] \delta = \delta(x_1)\ op \ldots op\ \delta(x_k)$ and $\overline{\Psi}_{j\delta} : \mathbf{Bool}^{n_j} \rightarrow \mathbf{Bool}^{n_j}$,

$\overline{\Psi}_{j\delta}(b_1, ..., b_{nj}) = ([\![op_{j_i} X_{j_i}]\!] (\delta \oslash [b_1 / x_1, ..., b_{nj} / x_{nj}]))_{1 \leq i \leq n_j}$

The local model-checking of a 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 Table 8. To every propositional variable Y

in the left-hand side of an equation and to 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.

Table 8
Translation of simple HMLR specifications into BESs

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

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

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

The $\mathbf{B}$ function is similar to other translations from modal equation systems to BESs [3,10,2,47,36]. $\mathbf{B}$ 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 Table 8), 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 resulting 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 terminate, 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, to keep a linear-time worst-case complexity, the computation results must be persistent between subsequent calls of the same solving routine [3].

3.2.1 Extended Boolean Graphs

Our resolution algorithm is easier to develop using a representation of BESs as *extended boolean graphs* [39], which are a slight generalization of the boolean graphs proposed in [2]. 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; $L : V \rightarrow \{\vee, \wedge\}$ 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 at which new outgoing 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 1 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 $\{x_0, x_5, x_8\}$.

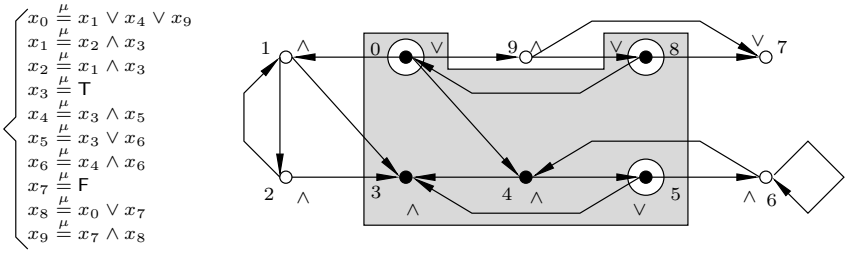

Fig. 1. A BES, its associated EBG, and a subgraph

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 Table 8).

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

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 using a μ-calculus formula, called *example formula*, interpreted over $\mathbf{G}$ [39]:

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

A variable x of the BES is true iff the vertex x satisfies EX in $\mathbf{G}$, noted $x \models_{\mathbf{G}} \text{EX}$. 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 $P_\wedge \wedge [\text{T}]\, \text{F}$). For the EBG in Figure 1, 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 1, 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* [39], 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 1, it is easy to see that the outlined subgraph is solution-closed: its frontier $\{x_0, x_5, x_8\}$ contains only $\vee$-vertices satisfying EX.

3.2.2 Local resolution algorithm

The SOLVE algorithm that we propose (see Figure 2) 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 the value computed in G.

SOLVE is similar in spirit with other graph-based local resolution algorithms like those of Andersen [2] and Vergauwen-Lewi [47]. However, since it implements the DFS iteratively, using an explicit stack and two nested while-loops, we believe that SOLVE is easier to understand than e.g., Andersen's algorithm, which uses a while-loop and two mutually recursive functions.

```
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)| 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. 2. Graph-based local resolution of a BES with sign μ

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 $y \models_{\mathbf{G}} \text{Ex}$ 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 $u \models_{\mathbf{G_A}} \text{Ex}$ 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 [3,10]: 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 $y \models_{\mathbf{G_A}} \text{Ex}$ ($c(y)$ is initialized to 1 for $\vee$-vertices and to $|E(y)|$ for $\wedge$-vertices). Thus, for every $y \in A$, $y \models_{\mathbf{G_A}} \text{Ex}$ if $c(y) = 0$.

Figure 3 shows the result of executing Solve for the variable x_0 and the Ebg in Figure 1 (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 $\mathbf{G_A}$.

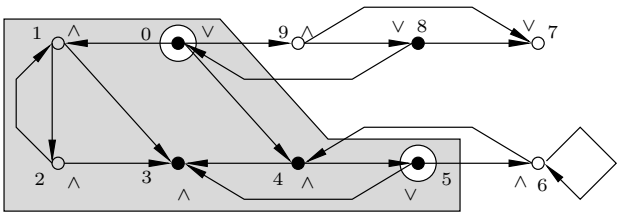

Fig. 3. A solution-closed subgraph computed by Solve

During the execution of Solve, the Dfs stack repeatedly takes one of the three forms outlined on Figure 4. 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 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[4], and thus after they are popped the stack takes again the form a). A proof of the partial correctness of the SOLVE algorithm can be found in Annex A.

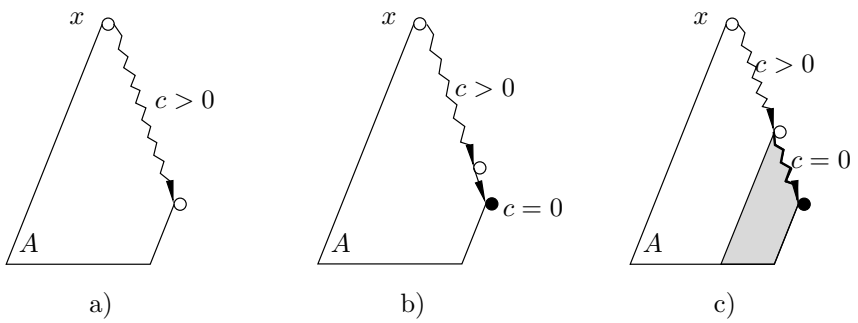

Fig. 4. Structure of the DFS stack during the execution of SOLVE

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 $x \models_{\mathbf{G_A}} \mathrm{EX}$ 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.

3.2.3 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. 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 $x \models_{\mathbf{G'}} \mathrm{EX}$ and *counterexample* if $x \not\models_{\mathbf{G'}} \mathrm{EX}$.

The SOLVE algorithm does not directly produce diagnostics; however, it can be easily coupled with the diagnostic generation algorithms proposed in [39].

[4] 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 4 c).

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 5 shows an example for the variable x_0 obtained by traversing again the solution-closed subgraph on Figure 3 previously computed by Solve.

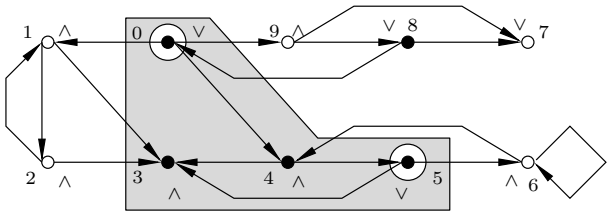

Fig. 5. An example for x_0

Since these diagnostic generation algorithms have a linear complexity in the size of the solution-closed subgraph they are executed upon [39], they affect neither the worst-case, nor the average-case complexity of Solve.

4 Implementation and applications

We used the model-checking method presented in Section 3 as a basis for developing the Evaluator 3.0 model-checker within the Cadp toolbox [19]. The tool has been constructed using the Open/Caesar environment [23], which provides a generic Api for on-the-fly exploration of (labeled) transition systems. As a consequence, Evaluator 3.0 can be used in conjunction with every compiler that is Open/Caesar–compliant (i.e., that implements a translation from its input language to the Open/Caesar Api), and particularly with the Caesar compiler [24] for Lotos [31].

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 for defining parameterized formulas and an inclusion mechanism for grouping 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 $\mathbf{E}[\varphi_{1\alpha_1}\mathbf{U}_{\alpha_2}\varphi_2]$ 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 $\text{EU_A_A}(\varphi_1, \alpha_1, \alpha_2, \varphi_2) = \mu Y.(\varphi_1 \wedge (\langle \alpha_2 \rangle \varphi_2 \vee \langle \alpha_1 \rangle Y))$. 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 obtained by using *wildcards* (regular expressions on character strings) as atomic predicates in action formulas. If actions of the LTS are represented as character strings (as it is currently the case with the OPEN/CAESAR API), this allows to specify a set of actions using a single predicate. For example, the wildcard `'SEND.*'` represents all LTS actions denoting the communication of 0 or more values on a gate `SEND`.

Table 9
Property patterns in regular alternation-free μ-calculus

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

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 [16] 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 [16] 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 [16]. Table 9 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. EVALUATOR 3.0 can be also used to search regular execution sequences in LTSs by checking basic PDL modalities: a transition sequence starting at the initial state and satisfying a regular formula β can be obtained either as an example for the $\langle\beta\rangle$ T formula, or as a counterexample for the $[\beta]$ F formula.

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 modeling 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 10). 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 $\texttt{Get}_{any}$ (abbreviations for `'Put.*'` and `'Get.*'` wildcards) denote the communication of an arbitrary message on gate `Put` and `Get`, respectively. Each 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 10
Properties of the Alternating Bit Protocol

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

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

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 previous version EVALUATOR 2.0, which accepts as input plain alternation-free μ-calculus formulas and implements the Fernandez-Mounier local boolean resolution algorithm [21]. All experiments have been performed on a Sparc Ultra 1 machine with 256 Mbytes of memory.

The results are shown in Table 11. 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 $n = 100$.

These results can be explained by a few observations. For invariant properties

that are true on the specification (e.g., P_4, P_5, and P_7), the speed-up obtained is roughly constant, since in this case both algorithms entirely explore the underlying boolean graph in order to decide the validity of the formulas. For properties that are either false (e.g., P_1 and P_6), or involve only a fragment of the LTS (e.g., P_2), the significant speed-ups obtained are mainly due to the different ways in which the two algorithms handle the portion of the boolean graph already explored. The SOLVE algorithm performs a single DFS traversal of the boolean graph, storing the values of the variables as soon as they have been computed, whereas the Fernandez-Mounier algorithm avoids as much as possible to store intermediate results, and therefore may perform multiple DFS traversals of the graph when the values of some variables need to be reused during the computation. It is worth noticing that, since both algorithms are based upon DFS traversals of the boolean graph (which induce DFS traversals of the LTS), changes in the order in which successor states are visited may strongly influence the time and memory required for the verification.

Table 11
Local model-checking statistics. a) EVALUATOR 3.0 (SOLVE algorithm); b) EVALUATOR 2.0 (Fernandez-Mounier algorithm)

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

In order to compare the performance of EVALUATOR 3.0 with other model-checking tools, we carried out the verification of a communication protocol, the i-PROTOCOL, which was proposed in [15] as a benchmark example for several widely-used model-checkers: COSPAN [28], MURφ [14], SMV [9], SPIN [29], and XMC [40]. The i-PROTOCOL is an optimized sliding window protocol, contained in the GNU UUCP package (available from the Free Software Foundation), designed to ensure ordered reliable duplex communication between two sites. At its lower interface, the i-PROTOCOL assumes an unreliable (lossy) packet-based FIFO connection. A problem with this protocol, which occurred when transmitting large data files over serial lines, was that, under certain message-loss conditions, the protocol will enter a livelock state from which no more data packet exchange can occur, ending up with a connection closing. The purpose of the benchmark proposed in [15] was to test the ability of different model-checkers to detect a livelock error in a real-life protocol.

Based on a description of the i-PROTOCOL in VPL (*Value Passing Language*) provided in [15], we developed a LOTOS specification of the protocol. It is worth noticing that, since VPL is an imperative language and LOTOS is a functional one, the specification obtained is not as suitable as it could be for the CAESAR compiler. We encoded the desired liveness property (which is very similar to the property P_6 shown in Table 10) as a regular alternation-free μ-calculus formula $[\mathrm{T}^*.\texttt{Send}]\,\mu Y.\,\langle \mathrm{T}\rangle\,\mathrm{T} \wedge [\neg \texttt{Recv}]\,Y$ stating that a message sent will be eventually received.

We checked this formula on the LOTOS specification by considering the same protocol configuration used in [15] (sliding window of size 2). The verification time needed by EVALUATOR 3.0 (about one second on a Sparc Ultra 1 machine with 256 Mbytes of memory) compares favourably with the time reported for XMC (about five seconds on a SGI IP25 Challenge machine with 1.9 Gbytes of memory), which was rated best on this example among the five model-checkers considered in [15]. The diagnostic generated by EVALUATOR 3.0 was a small counterexample sequence (15 transitions) containing a `Send` action and leading to a cycle without reaching a `Recv` action.

EVALUATOR 3.0 has been also used for the verification of other industrial applications: the SPLICE software coordination architecture for building distributed control systems [11,12], the GPRS mobile data packet radio service for GSM [35], an air traffic control system [41], a steam-boiler system [7], a truck lifting system [26], a distributed locker system [4], and a dynamic reconfiguration protocol for agent-based applications [1]. These experiments assessed the performance of the SOLVE algorithm and the usefulness of the diagnostic generation features.

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 [39], 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 protocol engineering toolbox [19] using the Open/Caesar environment [23].

The input language of Evaluator 3.0 allows to construct 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 [8], Actl [13], and a collection of generic property patterns proposed in [16] intended to facilitate the temporal logic specification activity.

Besides the applications cited in Sections 4.2 and 4.3, Evaluator 3.0 has been successfully experimented on various specifications of communication protocols and distributed systems described in Lotos, which are available in the Cadp distribution. The diagnostic generation features and the possibility of defining separate libraries of temporal operators appeared to be extremely useful in practice (in particular, for teaching purposes). Moreover, a connection between Evaluator 3.0 and the Orccad environment for robot controller design [42], 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 [30] 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 [20]. 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 [25] and full Lotos [31]. This can be done by translating data-based temporal logic formulas into parameterized boolean equation systems, which can be solved on-the-fly [38]. The implementation of these algorithms within the Cadp toolbox will require the extension of the Open/Caesar environment with data-handling facilities.

Acknowledgements

We are grateful to the anonymous referees for their valuable comments and suggestions for improvements. We also thank Frédéric Lang for his careful proofreading of the text, and Hubert Garavel for his support on the design and development of the EVALUATOR 3.0 model-checker. This work was partially supported by the INRIA Cooperative Research Action TOLERE directed by Alain Girault.

A Correctness proof of the SOLVE algorithm

This annex is devoted to the partial correctness proof of the SOLVE algorithm described in Section 3.2.2. For conciseness, we do not give all the details of the proof, but we focus instead on the essential properties ensuring the invariants of the while-loops of SOLVE. A complete formal proof, using e.g., Hoare's logic, could be constructed in a straightforward (but rather tedious) way.

A few additional notations are necessary. Let $G = (V, E, L, \emptyset)$ and $x \in V$ be the arguments of SOLVE and let $G_A = (A, \{(y, z) \mid y \in A \wedge z \in E(y) \cap A\}, L|_A, \{y \in A \mid p(y) < |E(y)|\})$ be the subgraph containing the vertices and edges that have been currently explored during the DFS traversal of G ($L|_A$ denotes the restriction to A of the labeling function $L : V \rightarrow \{\vee, \wedge\}$). Let $\mathbf{G}$ and $\mathbf{G_A}$ be the Kripke structures induced by G and G_A. The functional $\Phi^{\mathrm{Ex}}_{\mathbf{G_A}} : 2^S \rightarrow 2^S$, associated to the EX formula on $\mathbf{G_A}$, is defined as follows: $\Phi^{\mathrm{Ex}}_{\mathbf{G_A}}(U) = [\![(P_\vee \wedge \langle \mathrm{T} \rangle\, Y) \vee (P_\wedge \wedge [\mathrm{T}]\, Y)]\!]_{\mathbf{G_A}}\, [U/Y]$. For simplicity, we use the same symbol *stack* to denote the DFS stack and the set of vertices it contains.

The following lemma precises the invariants preserved by the while-loops of the SOLVE algorithm.

Lemma 1 (invariants of the while-loops of SOLVE)
The outer while-loop of SOLVE *preserves the main invariants I_1–I_3 given in Table A.1 and the auxiliary invariants J_1–J_3, K_1–K_3, and L_1–L_3 given in Table A.2. The inner while-loop of* SOLVE *preserves the invariants M_1–M_4 given in Table A.3.*

Table A.1
Main invariants of the while-loop of SOLVE

I_1	$\Phi^{\mathrm{Ex}}_{\mathbf{G_A}}(\{y \in A \mid c(y) = 0\}) \setminus stack = \{y \in A \setminus stack \mid c(y) = 0\}$
I_2	$\{y \in A \mid c(y) = 0\} \subseteq [\![\mathrm{Ex}]\!]_{\mathbf{G_A}}$
I_3	$\forall y \in A \setminus stack . p(y) < \lvert E(y)\rvert \Rightarrow (c(y) = 0 \wedge L(y) = \vee)$

Table A.2
Auxiliary invariants of the while-loop of SOLVE

J_1	$x \in \mathit{stack} \subseteq A$
J_2	$\forall y \in A.E(y) \cap A = \{z \in A \mid \exists 0 \leq i < p(y).z = (E(y))_i\}$
J_3	$\forall y \in \mathit{stack}$. if $y \to y_1 \to \cdots \to y_k \to \mathit{top}(\mathit{stack})$ is a sequence in G then $yy_1 \cdots y_k \mathit{top}(\mathit{stack})$ is a suffix of stack
K_1	$\mathit{stack} \neq \mathit{nil} \wedge c(\mathit{top}(\mathit{stack})) > 0 \Rightarrow \forall y \in \mathit{stack}.c(y) > 0$
K_2	$\mathit{stack} \neq \mathit{nil} \wedge c(\mathit{top}(\mathit{stack})) = 0 \wedge d(\mathit{top}(\mathit{stack})) \neq \emptyset \Rightarrow$ $d(\mathit{top}(\mathit{stack})) = \{\mathit{top}(\mathit{pop}(\mathit{stack}))\} \wedge \forall y \in \mathit{pop}(\mathit{stack}).c(y) = 0$
K_3	$\mathit{stack} \neq \mathit{nil} \wedge c(\mathit{top}(\mathit{stack})) = 0 \wedge d(\mathit{top}(\mathit{stack})) = \emptyset \Rightarrow$ the vertices in $\{y \in \mathit{stack} \mid c(y) = 0\}$ form a suffix of stack
L_1	$\forall y \in A.c(y) > 0 \Rightarrow d(y) = \{z \in A \mid y \in E(z)\}$
L_2	$\forall y \in A.$ $c(y) = \begin{cases} \lvert E(y)\rvert - \lvert\{z \in A \cap E(y) \mid c(z) = 0 \wedge y \notin d(z)\}\rvert & \text{if } L(y) = \wedge \\ \max(0, 1 - \lvert\{z \in A \cap E(y) \mid c(z) = 0 \wedge y \notin d(z)\}\rvert) & \text{if } L(y) = \vee \end{cases}$
L_3	$\forall y \in A.$ $c(y) = 0 \wedge (\mathit{stack} \neq \mathit{nil} \wedge d(\mathit{top}(\mathit{stack})) \neq \emptyset \Rightarrow y \neq \mathit{top}(\mathit{stack})) \Rightarrow d(y) = \emptyset$

Table A.3
Invariants of the inner while-loop of SOLVE

M_1	$\forall y \in B.c(y) = 0$
M_2	$\forall y \in \{z \in A \mid c(z) = 0\} \setminus B.d(y) = \emptyset$
M_3	$\forall y \in A.$ $c(y) = \begin{cases} \lvert E(y)\rvert - \lvert E(y) \cap (\{z \in A \mid c(z) = 0\} \setminus B)\rvert & \text{if } L(y) = \wedge \\ \max(0, 1 - \lvert E(y) \cap (\{z \in A \mid c(z) = 0\} \setminus B)\rvert) & \text{if } L(y) = \vee \end{cases}$
M_4	$\forall w \in B$. there exists a sequence $w \to w_1 \to \cdots \to w_k \to \mathit{top}(\mathit{stack})$ in G_A such that $c(w) = c(w_1) = \cdots = c(w_k) = c(\mathit{top}(\mathit{stack})) = 0$

Proof (sketch) Invariants I_1–I_3 are implied by the auxiliary invariants J_1–J_3, K_1–K_3, and L_1–L_3 of the outer while-loop and by the invariants M_1–M_4 of the inner while-loop.

Invariants J_1, J_2, and J_3 express general properties of the DFS traversal, independent of the SOLVE algorithm: J_1 states that the stack is included in A and contains x, which ensures that x is contained in G_A at the end of the while-loop; J_2 states that edges (y, z) of G_A are precisely the successor vertices of y numbered from 0 to $p(y) - 1$; and J_3 states that from any vertex y present on

the stack, the only sequence in G_A that leads from y to the stack top is the suffix of the stack beginning at y.

Invariants K_1, K_2, and K_3 characterize the stack structure during the DFS (they correspond to the three cases outlined in Figure 4): K_1 models case a), when every vertex on the stack is unstable; K_2 models case b), when only the top of the stack is stable, but this information has not yet been propagated to its predecessors; and K_3 models case c), when this propagation has terminated and all stable vertices present on the stack are adjacent to the top.

Invariants L_1, L_2, and L_3 express the relationship between the counters $c(y)$ and the backward dependencies $d(y)$ for all $y \in A$: L_1 states that for every unstable vertex y, all its predecessors in G_A are contained in $d(y)$; L_2 states that $c(y)$ counts all successors of y that remain to become stable in order to make y stable too; and L_3 states that every dependency to a stable vertex y has been taken into account and deleted from $d(y)$ (except for the top of the stack in the case characterized by K_2).

Invariants M_1, M_2, M_3, and M_4 express the basic properties of the inner while-loop, which performs (in the case characterized by K_2) the reevaluation of Ex on all vertices that depend upon the top of the stack: M_1, M_2, and M_3 state that all vertices contained in the work set B are stable, but this information has not yet been propagated through their backward dependencies; and M_4 states that from every vertex in B there is a sequence of stable vertices leading to the top of the DFS stack.

Invariants K_1–K_3 are implied by the general DFS properties J_1–J_3 and by the invariants M_1–M_4 of the inner while-loop. Invariants L_1–L_3 are implied by K_1–K_3 and M_1–M_4. Finally, the main invariants I_1–I_3 are implied by L_1–L_3 together with M_1–M_4. □

The following theorem states the partial correctness of the SOLVE algorithm.

Theorem 2 (partial correctness of SOLVE)
Upon termination of the SOLVE *procedure, the following conditions hold:*

(i) $[\![\mathrm{Ex}]\!]_{\mathbf{G_A}} = \{y \in A \mid c(y) = 0\}$
(ii) G_A *is a solution-closed subgraph of* G *containing* x

meaning that $x \models_{\mathbf{G}} \mathrm{Ex}$ *(i.e., the variable* x *is true in the* BES *denoted by* G*) iff* $c(x) = 0$.

Proof To show conditions (i) and (ii), we use the main invariants I_1, I_2, and I_3 of the outer while-loop of SOLVE, outlined in Table A.1. At the end of the

while-loop, these invariants ensure the properties below. Invariant I_1 implies that $\Phi^{\mathrm{Ex}}_{\mathbf{G_A}}(\{y \in A \mid c(y) = 0\}) = \{y \in A \mid c(y) = 0\}$, i.e., $\{y \in A \mid c(y) = 0\}$ is a fixed point of $\Phi^{\mathrm{Ex}}_{\mathbf{G_A}}$. By definition of $[\![\mathrm{Ex}]\!]_{\mathbf{G_A}}$ and invariant I_2, this implies property (i): $[\![\mathrm{Ex}]\!]_{\mathbf{G_A}} = \mu\Phi^{\mathrm{Ex}}_{\mathbf{G_A}} \subseteq \{y \in A \mid c(y) = 0\} \subseteq [\![\mathrm{Ex}]\!]_{\mathbf{G_A}}$. Invariant I_3 implies that the frontier of G_A contains only $\vee$-vertices satisfying Ex in $\mathbf{G_A}$: $\{y \in A \mid p(y) < |E(y)|\} \subseteq \{y \in A \mid c(y) = 0 \wedge L(y) = \vee\} = [\![P_\vee \wedge \mathrm{Ex}]\!]_{\mathbf{G_A}}$. Using the characterization of solution-closed EBGs given in [39], this implies condition (ii), i.e., G_A is a solution-closed subgraph of G. □

References

[1] M. Aguilar Cornejo, H. Garavel, R. Mateescu, N. de Palma, Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications, in: A. Laurentowski, J. Kosinski, Z. Mossurska, R. Ruchala (Eds.), *Proceedings of the 3rd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems DAIS'2001* (Krakow, Poland), pp. 229–242, Kluwer Academic Publishers, 2001. Full version available as INRIA Research Report RR-4222.

[2] H. R. Andersen, Model Checking and Boolean Graphs, *Theoretical Computer Science* 126 (1):3–30, 1994.

[3] A. Arnold, P. Crubillé, A Linear Algorithm to Solve Fixed-Point Equations on Transition Systems, *Information Processing Letters* 29:57–66, 1988.

[4] T. Arts, C. Benac Earle, Development of a Verified Erlang Program for Resource Locking, in: S. Gnesi, U. Ultes-Nitsche (Eds.), *Proceedings of the 6th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2001* (Paris, France), Univ. Paris 7 – LIAFA and INRIA Rhône-Alpes, pp. 131–145.

[5] H. Bekić, Definable Operations in General Algebras, and the Theory of Automata and Flowcharts, in: C. B. Jones (Ed.), *Programming Languages and their Definition*, Lecture Notes in Computer Science Vol. 177, pp. 30–55, Springer-Verlag, 1984.

[6] J. Bradfield, *Verifying Temporal Properties of Systems*, Birkhäuser, Berlin, 1992.

[7] P. J. F. Carreira, M. E. F. Costa, Automatically Verifying an Object-Oriented Specification of the Steam-Boiler System, in: S. Gnesi, I. Schieferdecker, A. Rennoch (Eds.), *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000* (Berlin, Germany), GMD Report 91, Berlin, 2000, pp. 345–360.

[8] E. M. Clarke, E. A. Emerson, 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, 1986.

[9] E. M. Clarke, K. McMillan, S. Campos, V. Hartonas-GarmHausen, Symbolic Model Checking, in: R. Alur, T. A. Henzinger (Eds.), *Proceedings of the 8th International Conference on Computer-Aided Verification CAV'96* (New Brunswick, NJ, USA), Lecture Notes in Computer Science Vol. 1102, pp. 419–422, Springer-Verlag, 1996.

[10] R. Cleaveland, B. Steffen, A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus, in: K. G. Larsen, A. Skou (Eds.), *Proceedings of 3rd Workshop on Computer Aided Verification CAV'91* (Aalborg, Denmark), Lecture Notes in Computer Science Vol. 575, pp. 48–58, Springer-Verlag, 1991.

[11] P. F. G. Dechering and I. A. van Langevelde, Towards Automated Verification of Splice in μCRL, Technical Report SEN-R0015, CWI, Amsterdam, 2000.

[12] P. F. G. Dechering and I. A. van Langevelde, On the Verification of Coordination, in: A. Porto and G.-C. Roman (Eds.), *Proceedings of the 4th International Conference on Coordination Models and Languages* (Limassol, Cyprus), Lecture Notes in Computer Science Vol. 1906, pp. 335–340, Springer-Verlag, 2000.

[13] R. D. Nicola, F. W. Vaandrager, Action versus State Based Logics for Transition Systems, in: I. Guessarian (Ed.), *Semantics of Systems of Concurrent Processes* (La Roche Posay, France), Lecture Notes in Computer Science Vol. 469, pp. 407–419, Springer-Verlag, 1990.

[14] D. L. Dill, The Murφ Verification System, in: R. Alur, T. A. Henzinger (Eds.), *Proceedings of the 8th International Conference on Computer-Aided Verification CAV'96* (New Brunswick, NJ, USA), Lecture Notes in Computer Science Vol. 1102, pp. 390–393, Springer-Verlag, 1996.

[15] Y. Dong, X. Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, D. S. Warren, Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools, in: R. Cleaveland (Ed.), *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'99* (Amsterdam, The Netherlands), Lecture Notes in Computer Science Vol. 1579, pp. 74–88, Springer-Verlag, 1999.

[16] M. B. Dwyer, G. S. Avrunin, J. 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), 1999.

[17] E. A. Emerson, C.-L. Lei, Efficient Model Checking in Fragments of the Propositional Mu-Calculus, in: *Proceedings of the 1st LICS*, pp. 267–278, 1986.

[18] A. Fantechi, S. Gnesi, F. Mazzanti, R. Pugliese, E. Tronci, A Symbolic Model Checker for ACTL, in: D. Hutter, W. Stephan, P. Traverso, M. Ullmann (Eds.), *Proceedings of the International Workshop on Current Trends in Applied Formal Methods FM-Trends'98* (Boppard, Germany), Lecture Notes in Computer Science Vol. 1641, Springer-Verlag, 1998.

[19] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, M. Sighireanu, CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox, in: R. Alur, T. A. Henzinger (Eds.), *Proceedings of the 8th International Conference on Computer-Aided Verification CAV'96* (New Brunswick, NJ, USA), Lecture Notes in Computer Science Vol. 1102, pp. 437–440, Springer-Verlag, 1996.

[20] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, C. Viho, Using On-the-Fly Verification Techniques for the Generation of Test Suites, in: R. Alur, T. A. Henzinger (Eds.), *Proceedings of the 8th International Conference on Computer-Aided Verification CAV'96* (New Brunswick, NJ, USA), Lecture Notes in Computer Science Vol. 1102, pp. 348–359, Springer-Verlag, 1996.

[21] J.-C. Fernandez, L. Mounier, A Local Checking Algorithm for Boolean Equation Systems, Rapport SPECTRE 95-07, VERIMAG, Grenoble, 1995.

[22] M. J. Fischer, R. E. Ladner, Propositional Dynamic Logic of Regular Programs, *Journal of Computer and System Sciences* 18:194–211, 1979.

[23] H. Garavel, OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing, in: B. Steffen (Ed.), *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98* (Lisbon, Portugal), Lecture Notes in Computer Science Vol. 1384, pp. 68–84, Springer-Verlag, 1998. Full version available as INRIA Research Report RR-3352.

[24] H. Garavel, J. Sifakis, Compilation and Verification of LOTOS Specifications, in: L. Logrippo, R. L. Probert, H. Ural (Eds.), *Proceedings of the 10th IFIP International Symposium on Protocol Specification, Testing and Verification* (Ottawa, Canada), 1990, pp. 379–394.

[25] J.-F. Groote, A. Ponse, The Syntax and Semantics of μCRL, Technical Report CS-R9076, CWI, Amsterdam, 1990.

[26] J.-F. Groote, J. Pang, A. Wouters, A Balancing Act: Analyzing a Distributed Lift System, in: S. Gnesi, U. Ultes-Nitsche (Eds.), *Proceedings of the 6th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2001* (Paris, France), Univ. Paris 7 – LIAFA and INRIA Rhône-Alpes, pp. 1–12. Full version available as CWI Report SEN-R0111.

[27] K. Hamaguchi, H. Hiraishi, S. Yajima, Branching Time Regular Temporal Logic for Model Checking with Linear Time Complexity, in: E. M. Clarke, R. P. Kurshan (Eds.), *Proceedings of the 2nd International Conference on Computer-Aided Verification CAV'90* (New Brunswick, NJ, USA), Lecture Notes in Computer Science Vol. 531, pp. 253–262, Springer-Verlag, 1990.

[28] R. H. Hardin, Z. Har'El, R. P. Kurshan, COSPAN, in: R. Alur, T. A. Henzinger (Eds.), *Proceedings of the 8th International Conference on Computer-Aided Verification CAV'96* (New Brunswick, NJ, USA), Lecture Notes in Computer Science Vol. 1102, pp. 423–427, Springer-Verlag, 1996.

[29] G. Holzmann, The Model Checker SPIN, *IEEE Transactions on Software Engineering* 23 (5):279–295, 1997.

[30] A. Ingolfsdottir, B. Steffen, Characteristic Formulae for Processes with Divergence, *Information and Computation* 110 (1):149–163, 1994.

[31] 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, 1988.

[32] D. Kozen, Results on the Propositional μ-calculus, *Theoretical Computer Science* 27:333–354, 1983.

[33] 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), Lecture Notes in Computer Science Vol. 299, pp. 215–230, Springer-Verlag, 1988.

[34] X. Liu, C. R. Ramakrishnan, S. A. Smolka, Fully Local and Efficient Evaluation of Alternating Fixed Points, in: B. Steffen (Ed.), *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98* (Lisbon, Portugal), Lecture Notes in Computer Science Vol. 1384, pp. 5–19, Springer-Verlag, 1998.

[35] L. Logrippo, L. Andriantsiferana, B. Ghribi, Prototyping and Formal Requirement Validation of GPRS: A Mobile Data Packet Radio Service for GSM, in: C.B. Weinstock, J. Rushby (Eds.), *Proceedings of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications DCCA-7* (San Jose, California, USA), 1999.

[36] A. Mader, *Verification of Modal Properties Using Boolean Equation Systems*, VERSAL 8, Bertz Verlag, Berlin, 1997.

[37] Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems, Vol. I (Specification)*, Springer-Verlag, 1992.

[38] R. Mateescu, Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus, in: A. Bossi, A. Cortesi, F. Levi (Eds.), *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation VMCAI'98* (Pisa, Italy), University Ca' Foscari of Venice, 1998.

[39] R. Mateescu, Efficient Diagnostic Generation for Boolean Equation Systems, in: S. Graf, M. Schwartzbach (Eds.), *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000* (Berlin, Germany), Lecture Notes in Computer Science Vol. 1785, pp. 251–265, Springer-Verlag, 2000. Full version available as INRIA Research Report RR-3861.

[40] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, D. S. Warren, Efficient Model Checking Using Tabled Resolution, in: *Proceedings of the 9th International Conference on Computer-Aided*

Verification CAV'97 (Haifa, Israel), Lecture Notes in Computer Science Vol. 1254, pp. 143–154, Springer-Verlag, 1997.

[41] M. Sage, C. Johnson, A Declarative Prototyping Environment for the Development of Multi-User Safety-Critical Systems, in: *Proceedings of the 17th International System Safety Conference ISSC'99* (Orlando, Florida, USA), System Safety Society, 1999.

[42] D. Simon, B. Espiau, K. Kapellos, R. Pissard-Gibollet, al., The ORCCAD Architecture, *International Journal of Robotics Research* 17 (4):338–359, 1998.

[43] R. Streett, Propositional Dynamic Logic of Looping and Converse, *Information and Control* 54:121–141, 1982.

[44] W. Thomas, Computation Tree Logic and Regular ω-languages, in: G. Rozenberg, J. W. de Bakker, W.-P. de Roever (Eds.), *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, Lecture Notes in Computer Science Vol. 354, pp. 690–713, Springer-Verlag, 1989.

[45] 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), pp. 408–418, IEEE Computer Society Press, 1995.

[46] M. Y. Vardi, P. Wolper, Reasoning about Infinite Computations, *Information and Computation* 115 (1):1–37, 1994.

[47] B. Vergauwen, J. Lewi, Efficient Local Correctness Checking for Single and Alternating Boolean Equation Systems, in: S. Abiteboul, E. Shamir (Eds.), *Proceedings of the 21st ICALP* (Vienna, Austria), Lecture Notes in Computer Science Vol. 820, pp. 304–315, Springer-Verlag, 1994.

[48] P. Wolper, Temporal Logic can be More Expressive, *Information and Control* 56 (1/2):72–99, 1983.

[49] P. Wolper, M. Y. Vardi, A. P. Sistla, Reasoning about Infinite Computation Paths, in: *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science FOCS'83* (Tucson, Arizona, USA), pp. 185–194, 1983.