A Model Checking Language for Concurrent Value-Passing Systems

Radu Mateescu, Damien Thivolle

A Model Checking Language for Concurrent Value-Passing Systems

Radu Mateescu and Damien Thivolle

INRIA Rhône-Alpes / VASY
655, avenue de l'Europe, F-38330 Montbonnot St Martin, France
{Radu.Mateescu,Damien.Thivolle}@inria.fr

Abstract. Modal μ-calculus is an expressive specification formalism for temporal properties of concurrent programs represented as Labeled Transition Systems (LTSs). However, its practical use is hampered by the complexity of the formulas, which makes the specification task difficult and error-prone. In this paper, we propose MCL (*Model Checking Language*), an enhancement of modal μ-calculus with high-level operators aimed at improving expressiveness and conciseness of formulas. The main MCL ingredients are parameterized fixed points, action patterns extracting data values from LTS actions, modalities on transition sequences described using extended regular expressions and programming language constructs, and an infinite looping operator specifying fairness. We also present a method for on-the-fly model checking of MCL formulas on finite LTSs, based on the local resolution of boolean equation systems, which has a linear-time complexity for alternation-free and fairness formulas. MCL is supported by the EVALUATOR 4.0 model checker developed within the CADP verification toolbox.

1 Introduction

Model checking [7] is an automatic, cost-effective method for verifying temporal properties of concurrent finite-state systems. In the action-based framework, where behaviours are represented as Labeled Transition Systems (LTSs), the modal μ-calculus ($L\mu$) [26, 38] provides a very expressive way of specifying properties. This fixed point-based logic subsumes virtually all other temporal logics defined in the literature; from this perspective, it can be seen as an "assembly language" for model checking on LTSs, similarly to the λ-calculus in the field of functional programming. The counterbalance of this expressiveness is the inherent complexity of $L\mu$ formulas, even for encoding relatively simple properties, which makes the practical usage of $L\mu$ difficult and error-prone, especially for non-expert users. In practice, higher-level formalisms are needed in order to facilitate the specification task and also to handle in a natural way the data values present in the LTSs generated from value-passing concurrent programs.

Towards this objective, classical temporal logics were extended with mechanisms inspired from regular languages and first-order logic. ETL [44] was the first extension of LTL [34] with regular grammars. BRTL [21] and ECTL* [41] are

extensions of CTL [7] and CTL* [12] with Büchi automata. Although they are strictly more powerful than the original ones, these enhanced logics are difficult to employ because of their complicated syntax. In practice, it appears that more concise and readable specifications can be obtained by using regular expressions, as illustrated by the SUGAR [6] extension of CTL and by regular $L\mu_1$ [32], which adds the modalities of test-free PDL [14] to $L\mu_1$, the alternation-free fragment of $L\mu$ [13]. Some extensions of CTL and LTL were further enriched with data and signal handling mechanisms, leading to specialized languages for hardware verification, such as PSL [23] and FORSPEC [3]. As regards $L\mu$, various combinations with first-order logic were proposed, especially in the field of symbolic verification [9, 19, 36] and of runtime verification [5]. However, no attempt of extending $L\mu$ both with regular expressions and data handling mechanisms was made so far in the framework of model checking for finite-state systems.

The experiences of using regular $L\mu_1$ for specifying temporal properties of industrial systems (ATM switches, asynchronous hardware, etc.) gave us a positive feedback about the gain in readability and conciseness of regular expressions w.r.t. fixed point operators. However, industrial users also formulated two requirements concerning the practical usage and the expressiveness of this logic:

- Temporal properties of value-passing systems must deal with the data values contained in the LTS, in order to avoid tedious updates of the properties for every configuration of the system under analysis (number of processes, values exchanged, etc.). Without parameterization mechanisms, temporal formulas may become prohibitively large because of operator instantiations capturing all relevant data values or expressing repetitions of transition sequences.
- Sometimes it is necessary to characterize precisely the presence of complex cycles (made of regular transition subsequences) in the LTS. In the absence of suitable fairness operators belonging to $L\mu_2$, users can detect complex cycles only by resorting to complicated schemes based on repetitive hiding and bisimulation minimization [4, 37].

In this paper, we attempt to fulfill these two requirements by proposing MCL (*Model Checking Language*), an extension of $L\mu$ with various operators aimed at improving the conciseness, readability, and expressiveness of temporal formulas. MCL combines data handling mechanisms (quantified variables and fixed point parameters), extended regular expressions, and constructs inspired from programming languages. All these features contribute to simplify the specification task, by drastically reducing the amount of fixed points and modalities in MCL formulas and allowing specifiers to focus their attention on the description of transition sequences. Fairness properties are expressed in MCL using the infinite looping operator of PDL-Δ [39], which enables a straightforward description of complex unfair sequences.

Besides improving the end-user language, our goal was also to maintain the complexity of verification as low as possible in order to deal with large LTSs. Therefore, as regards fixed point operators, we focused on the alternation-free fragment of MCL, which for dataless formulas coincides with $L\mu_1$ and takes advantage of its linear-time model checking complexity [8]. We reformulate the on-

the-fly verification of MCL formulas on LTSs as the local resolution of alternation-free boolean equation systems (BESs), by generalizing the classical procedures used for $L\mu_1$ [1, 42]. The infinite looping operator is expressible in $L\mu_2$, the $L\mu$ fragment of alternation depth 2 [13], whose model checking is quadratic; however, we show that this operator can be verified on-the-fly in linear time by proposing an enhanced BES resolution algorithm. This verification method is at the heart of the EVALUATOR 4.0 model checker that we developed within the CADP toolbox [17] using the generic OPEN/CÆSAR environment [15] for on-the-fly exploration of LTSs. As verification engine, the tool employs the generic CÆSAR_SOLVE library [31], which provides several linear-time local resolution algorithms for alternation-free BESs.

The paper is organized as follows. Section 2 defines the MCL language and illustrates its usage through various examples of properties. Section 3 describes the model checking method and the linear-time algorithm for handling the infinite looping operator. Section 4 presents the EVALUATOR 4.0 tool and its application for analyzing the SCSI-2 bus arbitration protocol [2]. Finally, Section 5 summarizes the results and indicates directions for future work.

2 Syntax and Semantics

MCL formulas are interpreted over LTSs of the form $M = \langle S, A, T, s_0 \rangle$, where S is the set of states, A is the 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 $s_1 \xrightarrow{a} s_2 \in T$ indicates that the system can move from state s_1 to state s_2 by performing action a. Actions in A are of the form $c\ v_1 ... v_n$, where c is a communication channel and $v_1, ..., v_n$ are the values exchanged during a handshake on c. The invisible action $\tau \notin A$ denotes an unobservable behaviour of the system. These LTSs are natural models for value-passing process algebras with early operational semantics, such as full CCS [33] and LOTOS [24].

2.1 Basic MCL: modal mu-calculus with data

A natural way of expressing properties about the values contained in LTS actions is to extend $L\mu$ with data variables, which can be quantified and used as parameters of fixed point operators. Our MCL language follows existing extension proposals [9, 19, 36] and enhances them by introducing higher-level constructs inspired from programming languages.

Basic MCL (see Fig. 1) consists of data expressions (e), action formulas (α), and state formulas (φ). Expressions are built from data variables $x \in \mathcal{X}$ and functions $f : T_1 \times \cdots \times T_n \rightarrow T$. Types bool and nat, equipped with the usual operations, are predefined, and all expressions are assumed to be well-typed. Action formulas are built from action patterns and boolean connectors. Action patterns inspect the structure of actions $c\ v_1 ... v_n$ by matching values v_i against expressions (clause "!e_i") or extracting and storing them in typed variables (clause "?x_i:T_i") exported to the enclosing formula.

Expressions:	$e ::= x \mid f(e_1, ..., e_n)$
Action formulas:	$\alpha ::= \{c\ !e_1 ... !e_n\} \mid \{c\ ?x_1{:}T_1 ... ?x_n{:}T_n\}$ $\mid\ \neg\alpha \mid \alpha_1 \vee \alpha_2$
State formulas:	$\varphi ::= e \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \langle\alpha\rangle\,\varphi$ $\mid$ exists $x_1{:}T_1, ..., x_n{:}T_n.\varphi \mid Y(e_1, ..., e_n)$ $\mid\ \mu Y(x_1{:}T_1{:=}e_1, ..., x_n{:}T_n{:=}e_n).\varphi$ $\mid$ let $x_1{:}T_1{:=}e_1, ..., x_n{:}T_n{:=}e_n$ in φ end let $\mid$ if φ_1 then φ'_1 elsif φ_2 then φ'_2 ... else φ'_n end if $\mid$ case e is $p_1 \to \varphi_1 \mid ... \mid p_n \to \varphi_n$ end case

Expressions:	$\lVert x \rVert\,\delta$	$=$	$\delta(x)$
	$\lVert f(e_1, ..., e_n) \rVert\,\delta$	$=$	$f(\lVert e_1 \rVert\,\delta, ..., \lVert e_n \rVert\,\delta)$
Action formulas:	$[\![\{c\ !e_1 ... !e_n\}]\!]\,\delta$	$=$	$\{c\ \lVert e_1 \rVert\,\delta ... \lVert e_n \rVert\,\delta\}$
	$[\![\{c\ ?x_1{:}T_1 ... ?x_n{:}T_n\}]\!]\,\delta$	$=$	$\{c\ v_1 ... v_n \mid \forall i \in [1, n].v_i \in T_i\}$
	$[\![\neg\alpha]\!]\,\delta$	$=$	$A \setminus [\![\alpha]\!]\,\delta$
	$[\![\alpha_1 \vee \alpha_2]\!]\,\delta$	$=$	$[\![\alpha_1]\!]\,\delta \cup [\![\alpha_2]\!]\,\delta$
	$env_{c\ v_1 ... v_n}(\{c\ ?x_1{:}T_1 ... ?x_n{:}T_n\})$	$=$	$[v_1/x_1, ..., v_n/x_n]$ if $\forall i \in [1, n].v_i \in T_i$
	$env_a(\alpha)$	$=$	$[\,]$ otherwise
State formulas:	$[\![e]\!]\,\rho\delta$	$=$	$\{s \in S \mid \lVert e \rVert\,\delta\}$
	$[\![\neg\varphi]\!]\,\rho\delta$	$=$	$S \setminus [\![\varphi]\!]\,\rho\delta$
	$[\![\varphi_1 \vee \varphi_2]\!]\,\rho\delta$	$=$	$[\![\varphi_1]\!]\,\rho\delta \cup [\![\varphi_2]\!]\,\rho\delta$
	$[\![\langle\alpha\rangle\,\varphi]\!]\,\rho\delta$	$=$	$\{s \in S \mid \exists s \xrightarrow{a} s'.a \in [\![\alpha]\!]\,\delta \wedge$ $s' \in [\![\varphi]\!]\,\rho(\delta \oslash env_a(\alpha))\}$
	$[\![\text{exists } x_1{:}T_1, ..., x_n{:}T_n.\varphi]\!]\,\rho\delta$	$=$	$\{s \in S \mid \exists v_1{:}T_1, ..., v_n{:}T_n.$ $s \in [\![\varphi]\!]\,\rho(\delta \oslash [v_1/x_1, ..., v_n/x_n])\}$
	$[\![Y(e_1, ..., e_n)]\!]\,\rho\delta$	$=$	$(\rho(Y))(\lVert e_1 \rVert\,\delta, ..., \lVert e_n \rVert\,\delta)$
	$[\![\mu Y(x_1{:}T_1{:=}e_1, ..., x_n{:}T_n{:=}e_n).\varphi]\!]\,\rho\delta$	$=$	$(\mu\Phi_{\rho\delta})(\lVert e_1 \rVert\,\delta, ..., \lVert e_n \rVert\,\delta)$
	where $\Phi_{\rho\delta} : (T_1 \times \cdots \times T_n \to 2^S) \to (T_1 \times \cdots \times T_n \to 2^S)$,		
	$(\Phi_{\rho\delta}(F))(v_1, ..., v_n) = [\![\varphi]\!]\,(\rho \oslash [F/Y])(\delta \oslash [v_1/x_1, ..., v_n/x_n])$		
	$\left[\!\!\left[\begin{array}{l}\text{let } x_1{:}T_1{:=}e_1, ...,\\ x_n{:}T_n{:=}e_n \text{ in } \varphi \text{ end let}\end{array}\right]\!\!\right]\rho\delta$	$=$	$[\![\varphi]\!]\,\rho(\delta \oslash [\lVert e_1 \rVert\,\delta/x_1, ..., \lVert e_n \rVert\,\delta/x_n])$
	$\left[\!\!\left[\begin{array}{l}\text{if } \varphi_1 \text{ then } \varphi'_1 \text{ elsif } \varphi_2 \text{ then } \varphi'_2\\ \quad ... \text{ else } \varphi'_n \text{ end if}\end{array}\right]\!\!\right]\rho\delta$	$=$	$\{s \in S \mid \text{if } s \in [\![\varphi_1]\!]\,\rho\delta \text{ then } s \in [\![\varphi'_1]\!]\,\rho\delta$ elsif $s \in [\![\varphi_2]\!]\,\rho\delta$ then $s \in [\![\varphi'_2]\!]\,\rho\delta$ ... else $s \in [\![\varphi'_n]\!]\,\rho\delta\}$
	$\left[\!\!\left[\begin{array}{l}\text{case } e \text{ is } p_1 \to \varphi_1 \mid\\ \quad ... \mid p_n \to \varphi_n \text{ end case}\end{array}\right]\!\!\right]\rho\delta$	$=$	if $\lVert e \rVert\,\delta :: p_1$ then $[\![\varphi_1]\!]\,\rho(\delta \oslash ext(\lVert e \rVert\,\delta, p_1))$ ... else $[\![\varphi_n]\!]\,\rho(\delta \oslash ext(\lVert e \rVert\,\delta, p_n))$

Fig. 1. Syntax (upper part) and semantics (lower part) of basic MCL operators

State formulas are built upon parameterized propositional variables $Y \in \mathcal{Y}$ and boolean expressions e by applying boolean connectors, modalities, quantifiers, and parameterized fixed point operators. Each variable Y denotes a function $F : T_1 \times \cdots \times T_n \rightarrow 2^S$ belonging to a set $\mathcal{F}$. Derived boolean operators ($\vee$, $\Rightarrow$, and $\Leftrightarrow$) and universal quantification (forall) are defined as usual in terms of $\neg$, $\wedge$, and exists. The necessity modality is the dual of possibility ($[\alpha]\,\varphi = \neg\,\langle\alpha\rangle\,\neg\varphi$) and the maximal fixed point operator is the dual of the minimal one ($\nu Y(...).\varphi = \neg\mu Y(...).\neg\varphi[\neg Y/Y]$, where $[\neg Y/Y]$ denotes the syntactic substitution of Y by $\neg Y$). Quantifiers may contain optional subdomain clauses "x:T among $\{e_1$ ... $e_2\}$" indicating that x takes values between e_1 and e_2. We allow quantification only on finite types equipped with a total order relation; existential and universal quantifiers are merely shorthand notations for (large) disjunctions and conjunctions parameterized by data values.

Fixed point formulas $\sigma Y(...).\varphi$, where $\sigma \in \{\mu, \nu\}$, are assumed to be *syntactically monotonic* [26], i.e., every free occurrence of Y in φ must fall in the scope of an even number of negations. For efficiency of model checking, we consider only *alternation-free* formulas, i.e., without mutual recursion between minimal and maximal fixed point variables, similarly to the $L\mu_1$ fragment [13].

Expressions e and action formulas α are interpreted in the context of a data environment $\delta : \mathcal{X} \rightarrow T_1 \cup ... \cup T_n$ assigning values to all free variables occurring in e and α (the environment $\delta \oslash [v/x]$ is identical to δ except for variable x, which is assigned value v). State formulas φ are interpreted also in the context of propositional environments $\rho : \mathcal{Y} \rightarrow \mathcal{F}$, which assign functions to all free propositional variables occurring in φ. The parameterized fixed point operators $\sigma Y(x_1{:}T_1{:=}e_1, ..., x_n{:}T_n{:=}e_n).\varphi$ represent both the definition and the call (with the values of $e_1, ..., e_n$ as arguments) of functions $F : T_1 \times \cdots \times T_n \rightarrow 2^S$ defined as the corresponding fixed points of monotonic functionals over $T_1 \times \cdots \times T_n \rightarrow 2^S$.

To facilitate the handling of data values, we introduce the "let", "if", and "case" operators, inspired from functional programming languages. The branches of "if" and "case" formulas must be exhaustive in order to avoid exceptions and the formulas φ_i used as branch conditions in "if" must not contain free propositional variables in order to preserve syntactic monotonicity. Patterns in "case" formulas are of the form "x:T" or "$f(p_1, ..., p_n)$", where f is a constructor of the type of e. Variables defined in patterns p_i are visible in the state formulas φ_i of their corresponding branches. The predicate "v::p" indicates whether value v matches p or not, and $ext(v, p)$ denotes the data environment initializing all variables defined in p with their values extracted from v by pattern-matching.

Action patterns have additional features (not shown in Fig. 1): the two kinds of clauses can be mixed; the wildcard clause "any" matches a value of any type; for dataless actions, brackets can be omitted; and an optional guard "where e" ending the clause list indicates that the action pattern matches an action iff the boolean expression e (which can refer to the variables declared in the clauses "?x:T" of the action pattern) evaluates to true.

A state satisfies a closed formula φ (notation $s \models \varphi$) iff $s \in [\![\varphi]\!]$. An LTS $M = \langle S, A, T, s_0\rangle$ satisfies a closed formula φ (notation $M \models \varphi$) iff $s_0 \models \varphi$. Basic

Table 1. Temporal properties formulated using basic MCL operators (upper part) and extended regular operators (lower part). φ_1 and φ_2 involve action patterns, quantifiers, and boolean expressions. φ_3 counts using a parameterized fixed point and φ_4 simulates a simple pushdown automaton for syntactic analysis.

	MCL formula	Meaning
φ_1	$[\mathsf{true}^*.\{open\ ?i{:}\mathsf{nat}\}.(\neg\{close\ !i\})^*.$ $\{open\ ?j{:}\mathsf{nat}\}]\ (i = j)$	Mutual exclusion between processes i and j.
φ_2	$[\{bcast\ ?msg{:}\mathsf{nat}\}]$ $\mathsf{forall}\ addr{:}\mathsf{nat}.$ $\mu Y.(\langle\mathsf{true}\rangle\,\mathsf{true} \wedge [\neg\{recv\ !msg\ !addr\}]\,Y)$	Inevitable reception of a broadcasted message msg at all its destinations $addr$.
φ_3	$\nu Y(c{:}\mathsf{nat}{:=}0).\mathsf{if}\ c = 3\ \mathsf{then}\ \langle\mathsf{true}^*.resp\rangle\,\mathsf{true}$ $\mathsf{else}\ [req_1 \vee req_2 \vee req_3]\,Y(c+1)$ $\mathsf{end\ if}$	Potential response after three requests occurring in any order.
φ_4	$\nu X(n{:}\mathsf{nat}{:=}0).([open\_par]\,X(n+1) \wedge$ $[close\_par]\,(n > 0 \wedge X(n-1)) \wedge$ $[eof]\,(n = 0) \wedge$ $[\neg(open\_par \vee close\_par \vee eof)]\,X(n))$	No sequence of transitions (tokens) can reach an eof without having well-balanced parentheses.
φ_5	$[\mathsf{true}^*.((\neg output)^*.input)\{n+1\}]\,\mathsf{false}$	Safety of a n-place buffer.
φ_6	$[\{level\ ?l{:}\mathsf{nat}\}]\,((l > max)\ \Rightarrow$ $[(\neg alarm)\{16\}]\,\mathsf{false} \wedge$ $[(\neg alarm)\{0\ ...\ 15\}]\,\langle\mathsf{true}\rangle\,\mathsf{true})$	Inevitable alarm at most 15 transitions (ticks) after a level reaches a threshold.
φ_7	$[\mathsf{true}^*.\{ask\ ?i{:}\mathsf{nat}\}]$ $\langle(\neg\{get\ !i\})^*.\{get\ ?j{:}\mathsf{nat\ where}\ j \neq i\}\rangle\,@$	Starvation of process i in favor of another process j.
φ_8	$\langle\mathsf{true}^+.\mathsf{if}\ \neg p_{final}\ \mathsf{then\ false\ end\ if}\rangle@$	Acceptance condition in a Büchi automaton (p_{final} denotes the repeated states).

MCL allows to express naturally temporal properties involving data values, as illustrated in Table 1. Other data-based properties will be shown in Section 4.

2.2 Extended regular operators

Besides the data handling operators of basic MCL, which bring the benefits of parameterization, another kind of useful extension is the ability of specifying transition sequences using regular expressions [6]; in the context of $L\mu$, this can be done naturally by plugging regular expressions inside modalities, similarly to PDL [14]. Although these modalities can be translated into $L\mu_1$ [13], in practice they are much more concise and readable than their fixed point counterparts [32].

The regular formulas (β) we propose in MCL for specifying transition sequences are built from action formulas and various operators stemming from extended regular expressions and programming languages (see Tab. 2). For conciseness, we define the meaning of β formulas by giving their translations to basic MCL when they occur in $\langle\ \rangle$ modalities (dual translations hold for $[\ ]$ modalities). The counter-based iteration operators are inspired from the extended regular expressions implemented in string searching tools like the `egrep` utility available on

Table 2. Syntax and semantics of (a subset of) MCL extended regular operators

Syntax	Meaning	Translation
$\langle \mathsf{nil} \rangle\, \varphi$	Empty seq.	φ
$\langle \beta_1.\beta_2 \rangle\, \varphi$	Concatenation	$\langle \beta_1 \rangle \langle \beta_2 \rangle\, \varphi$
$\langle \beta_1 \mid \beta_2 \rangle\, \varphi$	Choice	$\langle \beta_1 \rangle\, \varphi \vee \langle \beta_2 \rangle\, \varphi$
$\langle \beta? \rangle\, \varphi$	Option	$\varphi \vee \langle \beta \rangle\, \varphi$
$\langle \beta^* \rangle\, \varphi$	Iter. ≥ 0 times	$\mu Y.(\varphi \vee \langle \beta \rangle\, Y)$
$\langle \beta^+ \rangle \varphi$	Iter. ≥ 1 times	$\mu Y.\, \langle \beta \rangle\, (\varphi \vee Y)$
$\langle \beta\{e\} \rangle\, \varphi$	Iteration e times	$\mu Y(c{:}\mathsf{nat}{:=}e).$ $\quad$ if $c > 0$ then $\langle \beta \rangle\, Y(c-1)$ $\quad\quad$ else φ end if
$\langle \beta\{e_1 \ldots e_2\} \rangle\, \varphi$	Iteration $e_1 \leq e_2$ times	$\mu Y(c_1{:}\mathsf{nat}{:=}e_1, c_2{:}\mathsf{nat}{:=}e_2{-}e_1).$ $\quad$ if $c_1 > 0$ then $\langle \beta \rangle\, Y(c_1-1, c_2)$ $\quad\quad$ elsif $c_2 > 0$ then $\quad\quad\quad \varphi \vee \langle \beta \rangle\, Y(c_1, c_2-1)$ $\quad\quad$ else φ end if
$\langle$ let $x_1{:}T_1{:=}e_1, \ldots, x_n{:}T_n{:=}e_n$ in β end let$\rangle\, \varphi$	Variable definition	let $x_1{:}T_1{:=}e_1, \ldots, x_n{:}T_n{:=}e_n$ in $\langle \beta \rangle\, \varphi$ end let
$\langle$ if φ_1 then β_1 $\quad$ elsif φ_2 then β_2 ... else β_n end if$\rangle\, \varphi$	Conditional	if φ_1 then $\langle \beta_1 \rangle\, \varphi$ $\quad$ elsif φ_2 then $\langle \beta_2 \rangle\, \varphi$...else $\langle \beta_n \rangle\, \varphi$ end if
$\langle$ case e is $\quad p_1 \rightarrow \beta_1 \mid \ldots \mid p_n \rightarrow \beta_n$ end case$\rangle\, \varphi$	Selection	case e is $\quad p_1 \rightarrow \langle \beta_1 \rangle\, \varphi \mid \ldots \mid p_n \rightarrow \langle \beta_n \rangle\, \varphi$ end case
$\langle$ while φ_1 do β end while$\rangle\, \varphi_2$	Initial test loop	$\mu Y.$if φ_1 then $\langle \beta \rangle\, Y$ else φ_2 end if

UNIX systems. These operators turn out to be as useful for specifying transition sequences as their `egrep` counterparts are for describing character strings (see, e.g., formulas φ_5 and φ_6 in Tab. 1). However, when one must handle the data values contained in actions and characterize the intermediate states occurring on a sequence, more sophisticated operators become necessary.

The "if" operator generalizes the testing operator "$?\varphi$" of PDL [14], which allows to specify a formula φ about an intermediate state of a sequence denoted by a regular formula. The "$?\varphi$" operator is formulated in MCL as if $\neg\varphi$ then false end if. The "let" and "case" operators are the sequence counterparts of the corresponding state operators. Note that exhaustiveness of branches in the "if" and "case" regular formulas is not mandatory: if none of the branch conditions is satisfied, the formula denotes the empty sequence, exactly as in sequential programming languages. The "while" iteration operator specifies repetitions of subsequences driven by their source states. Originally, the "if" and "while" operators were introduced in *well-structured* PDL [20], a syntactic extension of PDL intended to enforce a disciplined use of the testing operator.

The regular modalities defined in Table 2 can deal only with finite transition sequences and thus can specify only simple fairness properties, such as the fair reachability [35] of an action a, described as $[(\neg a)^*]\,\langle(\neg a)^*.a\rangle\,\mathsf{true}$. To specify more complex fairness properties, we use the infinite looping operator $\Delta\beta$ of PDL-Δ [39], noted $\langle\beta\rangle$ @ in MCL, which states the existence of an infinite (unfair) sequence made by concatenating subsequences satisfying β (see formula φ_7 in Tab. 1). This operator is equivalent to the $\nu Y.\,\langle\beta\rangle\,Y$ formula [13]; by expanding β using the rules in Table 2, the resulting formula belongs to $L\mu_2$ or $L\mu_1$, depending whether β contains iteration operators or not. The $\langle\beta\rangle$ @ operator captures the Büchi acceptance condition (see formula φ_8 in Tab. 1), unexpressible in $L\mu_1$.

Expressiveness: The dataless fragment of MCL (i.e., with no occurrences of data variables) contains the operators of $L\mu_1$, the regular modalities of PDL (embedded in $L\mu_1$ [13]), and the infinite looping operator, which belongs to $L\mu_2$ [13]. This fragment strictly includes PDL-Δ, which in turn subsumes LTL (see formula φ_8 in Tab. 1) and CTL* [43]. Data variables do not, strictly speaking, increase expressiveness: since we work on finite LTSs, all possible instances of data variables or fixed point parameters related to the LTS could be expanded statically. This is however uncompatible with the on-the-fly model checking approach, which does not assume an *a priori* knowledge of the entire LTS.

3 Model Checking Method

We are interested in the on-the-fly model checking of an alternation-free MCL formula φ on an LTS $M = \langle S, A, T, s_0\rangle$, which consists in determining whether s_0 satisfies φ or not by exploring T in a forward manner starting at s_0. We proceed by generalizing the method used for model checking $L\mu_1$ formulas [1, 8] and their extensions with PDL regular modalities [32]: the verification problem is reformulated as the local resolution of a boolean equation system (BES) [28], which is performed using specialized algorithms such as those in [31].

3.1 Translation into parameterized BESs

The reformulation of the verification problem roughly consists of four steps, illustrated in the table below. The MCL formula φ serving as example states that every number p inserted into an empty 5-place buffer will be potentially delivered after 4 internal transitions denoting the moves of p between contiguous buffer cells. The first three steps transform φ syntactically, and the fourth one involves semantic information coming from the LTS.

1. The normalization step inserts new propositional variables at appropriate places in order to capture all occurrences of "hidden" fixed points underlying regular modalities with iterations (e.g., νY_1) and to ensure that every data variable occuring free in a subformula must be a parameter of the propositional variable dominating the subformula (e.g., $\mu Z_1(q{:}\mathsf{nat}{:=}p)$).

2. The translation to parameterized PDL with recursion (PDLR) [32] brings the formula to an equivalent equational form. Here we focus on alternation-free PDLR systems, without cyclic dependencies between equation blocks. Every

fixed point subformula induces an equation, which (due to normalization) is self-contained w.r.t. its data parameters.

3. The translation to parameterized HML with recursion (HMLR) [27] simplifies the modalities by expanding the regular formulas according to their semantics given in Table 2. Duplication of subformulas is avoided by introducing new equations, in such a way that the size of the HMLR system remains linear w.r.t. the initial MCL formula.

4. The final step makes a kind of product between the HMLR system and the LTS, producing a parameterized BES (PBES) [29] in which a boolean variable $Y_s(v_1, ..., v_n)$ is true iff state s satisfies $Y(v_1, ..., v_n)$. The evaluation of the boolean formulas in the right-hand sides allows to traverse the LTS transitions in a forward way, suitable for on-the-fly verification.

MCL formula	$[\mathsf{true}^*.\{put\ ?p{:}\mathsf{nat}\}]\ \langle\tau\{4\}.\{get\ !p\}\rangle\ \mathsf{true}$
Normalized formula	$\nu Y_1.\ [\mathsf{true}^*.\{put\ ?p{:}\mathsf{nat}\}]\ \mu Z_1(q{:}\mathsf{nat}{:=}p).\ \langle\tau\{4\}.\{get\ !q\}\rangle\ \mathsf{true}$
PDLR system	$\left\{ Y_1 \stackrel{\nu}{=} [\mathsf{true}^*.\{put\ ?p{:}\mathsf{nat}\}]\ Z_1(p) \right\}$ $\left\{ Z_1(q{:}\mathsf{nat}) \stackrel{\mu}{=} \langle\tau\{4\}.\{get\ !q\}\rangle\ \mathsf{true} \right\}$
HMLR system	$\left\{ Y_1 \stackrel{\nu}{=} [\mathsf{true}^*]\ [\{put\ ?p{:}\mathsf{nat}\}]\ Z_1(p) \right\}$ $\left\{ Z_1(q{:}\mathsf{nat}) \stackrel{\mu}{=} \langle\tau\{4\}\rangle\ \langle\{get\ !q\}\rangle\ \mathsf{true} \right\}$
	$\left\{ Y_1 \stackrel{\nu}{=} [\{put\ ?p{:}\mathsf{nat}\}]\ Z_1(p) \wedge [\mathsf{true}]\ Y_1 \right\}$ $\left\{ \begin{array}{l} Z_1(q{:}\mathsf{nat}) \stackrel{\mu}{=} Z_2(q, 4) \\ Z_2(q, c{:}\mathsf{nat}) \stackrel{\mu}{=} \mathsf{if}\ c > 0\ \mathsf{then}\ \langle\tau\rangle\ Z_2(q, c-1) \\ \qquad\qquad \mathsf{else}\ \langle\{get\ !q\}\rangle\ \mathsf{true}\ \mathsf{end\ if} \end{array} \right\}$
PBES	$\left\{ Y_{1s} \stackrel{\nu}{=} \bigwedge_{s \stackrel{put\ m}{\rightarrow} s'} Z_{1s'}(m) \wedge \bigwedge_{s \rightarrow s'} Y_{1s'} \right\}_{s \in S}$ $\left\{ \begin{array}{l} Z_{1s}(q{:}\mathsf{nat}) \stackrel{\mu}{=} Z_{2s}(q, 4) \\ Z_{2s}(q, c{:}\mathsf{nat}) \stackrel{\mu}{=} \mathsf{if}\ c > 0\ \mathsf{then}\ \bigvee_{s \stackrel{\tau}{\rightarrow} s'} Z_{2s'}(q, c-1) \\ \qquad\qquad \mathsf{else}\ \bigvee_{s \stackrel{get\ q}{\rightarrow} s'} \mathsf{true}\ \mathsf{end\ if} \end{array} \right\}_{s \in S}$

The evaluation of an MCL formula on the initial state s_0 of an LTS is reduced to the resolution of a variable instance $Y_{s_0}(v_1, ..., v_n)$ defined by the first equation block of the corresponding PBES. This is carried out by expanding the PBES incrementally, starting at $Y_{s_0}(v_1, ..., v_n)$ and evaluating the formula in the right-hand side of its equation, which in turn will generate new variable instances, and so on. If the number of generated instances is finite, the expanded PBES portion is converted into a plain BES by associating a boolean variable $Y_{s,v_1,...,v_n}$ to each variable instance $Y_s(v_1, ..., v_n)$, based on the isomorphism of the lattices $(T_1 \times \cdots \times T_n \rightarrow \mathsf{bool})^{|S|}$ and $\mathsf{bool}^{|T_1| \cdots |T_n| \cdot |S|}$. Then, the value of $Y_{s_0,v_1,...,v_n}$ is obtained by locally solving the resulting BES using the linear-time algorithms of [31]. The incremental expansion of the PBES to a plain BES and the local resolution are performed simultaneously, since both of them rely upon a forward exploration of the dependencies between (instances of) boolean variables.

3.2 Handling of the infinite looping operator

When β contains iterations, the $\langle\beta\rangle$ @ operator corresponds to a $L\mu_2$ formula; although we cannot directly use alternation-free BES resolution, we can still devise a linear-time algorithm for evaluating it on an LTS. The PDLR system equivalent to $\langle\beta\rangle$ @ contains two mutually recursive equation blocks $\{Y \stackrel{\nu}{=} Z\}$ and $\{Z \stackrel{\mu}{=} \langle\beta\rangle\, Y\}$. The BES obtained by applying the translation given in Section 3.1 has the form $\{Y_s \stackrel{\nu}{=} Z_s\}$, $\{Z_s \stackrel{\mu}{=} \vee_{s \xrightarrow{\beta} s'} Y_{s'}\}$. The shorthand notation $\vee_{s \xrightarrow{\beta} s'}$ means the existence of a sequence relating s and s' and satisfying β; in general, this is further expanded into several disjunctive equations, e.g., the block $\{Z_s \stackrel{\mu}{=} \vee_{s \xrightarrow{(a^* \cdot b)^* \cdot c} s'} Y_{s'}\}$ becomes $\{Z_s \stackrel{\mu}{=} \vee_{s \xrightarrow{c} s'} Y_{s'} \vee W_s, W_s \stackrel{\mu}{=} \vee_{s \xrightarrow{b} s'} Z_{s'} \vee \vee_{s \xrightarrow{a} s'} W_{s'}\}$. To solve such a BES, we abusively merge its two blocks into a single disjunctive μ-block $B = \{Y_s \stackrel{\mu}{=} Z_s, Z_s \stackrel{\mu}{=} \vee_{s \xrightarrow{\beta} s'} Y_{s'}\}$ and take care to preserve the original semantics during local resolution. A state s satisfies $\langle\beta\rangle$ @ iff it has an outgoing infinite sequence made of subsequences satisfying β. If there is no such sequence going out of s, then the formula is false, which is also the value of Y_s obtained by solving B. Otherwise, the infinite sequence ends with a cycle (because the LTS is finite) going through some variable $Y_{s'}$; to force Y_s to true, it is sufficient to detect the cycle and set $Y_{s'}$ to true, which will propagate back to Y_s.

The local resolution algorithm A4 [31] solves a disjunctive BES by performing a depth-first search (DFS) of the associated *boolean graph* [1], which encodes the dependencies between boolean variables. The DFS starts at the vertex (variable) of interest x and, when it encounters a true constant, it propagates it back to x via the disjunctive variables present on the DFS stack, which become true as well. Furthermore, all visited variables that may reach the true constant must also be set to true, in order to avoid multiple traversals of the boolean graph during subsequent invocations of A4 (this happens when the subformula from which the block was generated is nested within other temporal operators) and to keep a linear-time complexity for the overall resolution. Since these variables belong to the partially explored strongly connected components (SCCs) covering the DFS stack, A4 also performs SCC detection using Tarjan's algorithm [40].

Algorithm A4$_{cyc}$ (see Fig. 2) extends A4 with the ability to detect cycles going through certain marked variables (such as $Y_{s'}$ above), indicated by a predicate *marked*. When such a cycle is detected, the marked variable becomes true, and its value will propagate back to x via disjunctive variables, exactly like an ordinary true constant. A simple way to detect these cycles is to check, every time a SCC is identified, whether it contains such a variable or not. A more efficient solution is to do the check only when traversing a cycle-closing edge (lines 20–23), i.e., a "back" or a "cross" edge in the DFS terminology [40]. To avoid searching the DFS stack for marked variables, we use an additional $stack_2$, which contains all marked variables present on the DFS stack and evolves synchronously with it. Then, it is sufficient to check that the target variable of a cycle-closing edge has a "lowlink" number [40] smaller than the DFS number of the variable on top of $stack_2$. Thus, cycles containing marked variables are always detected before the exploration of the last encountered SCC is completed.

```
1. var A : 2^V ; k : nat ; stack : V* ; stack_2 : V* ;
2. A := ∅ ; k := 0 ; stack := nil ;
3. function A4_cyc (x : V, (V, E, L), marked : V → bool) : bool is
4.    var v, stable : V → bool ; n, p, low : V → nat ;
5.        y, z : V ; val : bool ;
6.    if |E(x)| = 0 then
7.       if L(x) = ∨ then v(x) := false else v(x) := true end if ;
8.       stable(x) := true
9.    else
10.      v(x) := false;
11.      stable(x) := false
12.   end if ;
13.   p(x) := 0 ; n(x) := k ; k := k + 1 ; low(x) := n(x) ;
14.   A := A ∪ {x} ; stack := push(x, stack) ;
15.   if marked(x) then stack_2 := push(stack_2) end if ;
16.   while p(x) < |E(x)| do
17.      y := (E(x))_p(x) ;
18.      if y ∈ A then
19.         val := v(y) ;
20.         if ¬stable(y) ∧ n(y) < n(x) then
21.            low(x) := min(low(x), n(y)) ;
22.            if low(x) ≤ n(top(stack_2)) then val := true end if
23.         end if
24.      else
25.         val := A4_cyc (y, (V, E, L))
26.         low(x) := min(low(x), low(y))
27.      end if ;
28.      if val then
29.         v(x) := true ; stable(x) := true ;
30.         p(x) := |E(x)|
31.      else
32.         p(x) := p(x) + 1
33.      end if
34.   end while ;
35.   if v(x) ∨ low(x) = n(x) then
36.      repeat
37.         z := top(stack) ; v(z) := v(x) ;
38.         stable(z) := true ; stack := pop(stack)
39.      until z = x
40.   end if ;
41.   if x = top(stack_2) then stack_2 := pop(stack_2) end if ;
42.   return v(x)
43. end
```

Fig. 2. Local resolution of a disjunctive μ-block with marked cycle detection

Complexity of model checking: Algorithm A4$_{cyc}$ runs in $O(|V|+|E|)$ time and $O(|V|)$ memory, where V is the set of boolean graph vertices (boolean variables) and E is the set of dependencies between them (boolean operators). Since A4 can handle PDL [31], it provides together with A4$_{cyc}$ a linear-time on-the-fly model checking procedure for PDL-Δ, improving over the classical quadratic procedure obtained by translating PDL-Δ to $L\mu_2$ [13]. Given that $\langle\beta\rangle$ @ captures the Büchi acceptance condition, A4$_{cyc}$ could also serve as verification back-end for LTL; the SCC detection (which is not needed for LTL model checking [22]) is necessary for ensuring a linear-time complexity when $\langle\beta\rangle$ @ occurs nested within branching-time operators and is therefore evaluated multiple times.

Dataless formulas of MCL are evaluated with a time and space complexity $O(|\varphi| \cdot (|S| + |T|))$ using BES resolution [31]. The operators of CTL and PDL-Δ, which cover the quasi-totality of practical needs, are evaluated with a space complexity $O(|\varphi| \cdot |S|)$ using A4 and A4$_{cyc}$. Data variables of infinite types may lead to divergence of model checking, because the number of boolean variable instances produced by expansion from PBESs to BESs can be unbounded. Therefore, parameterized fixed points should be used with the same care as recursive functions in programming languages (note however that cycles $Y_s(v_1, ..., v_n) \rightarrow \cdots \rightarrow Y_s(v_1, ..., v_n)$ do not harm, since BES resolution algorithms can handle cyclic dependencies between variables). The evaluation of all extended regular operators given in Table 2 is guaranteed to converge, because their expansion to BESs always creates a finite number of variable instances, bounded by the values of iteration counters and/or the number of LTS states.

4 Implementation and Use

The model checkers Evaluator 3.x and 4.0: A verification method similar to that given in Section 3, but restricted to dataless formulas, is at the core of the EVALUATOR 3.x model checker [31, 32] of CADP [17], which evaluates formulas of regular $L\mu_1$ on LTSs on-the-fly. The tool is based on the generic LTS exploration API defined by OPEN/CÆSAR [15] and therefore is language-independent. The implicit BES produced by reformulating the verification problem is solved on-the-fly using the local resolution algorithms of the generic CÆSAR_SOLVE library [31]. These algorithms rely upon various exploration strategies of boolean graphs: plain DFS, optimized DFS for the memory-efficient resolution of disjunctive/conjunctive or acyclic BESs, breadth-first search (BFS), etc. The tool is also able to generate examples and counterexamples (LTS portions explaining the verification result) using the BES approach proposed in [30]. To facilitate the specification task, derived temporal operators can be defined as macros parameterized by subformulas and grouped into reusable libraries, several of which are currently available (defining CTL [7], Action-based CTL [10], and the property patterns proposed in [11]). EVALUATOR 3.x served to validate more than 30 industrial case-studies over the last 7 years[1], and is currently used by BULL, STMICROELECTRONICS, and CEA/LETI for checking asynchronous hardware [37].

[1] See `http://www.inrialpes.fr/vasy/cadp/case-studies`

The verification method described in Section 3 is at the core of the new version EVALUATOR 4.0 (38 000 lines of code) that we developed within CADP. The new tool[2] brings two major enhancements w.r.t. its previous versions 3.x:

- The MCL language presented in Section 2 (which is a conservative extension of regular $L\mu_1$, the input language of versions 3.x) is now accepted as input, thus allowing to express temporal properties involving data. The presence of data parameters significantly reduces the size of temporal specifications, by avoiding tedious repetitions of formulas caused by instantiations of parameters with values contained in the LTS actions.
- The infinite looping operator $\langle ... \rangle$ @ is now fully implemented (versions 3.x only accepted iteration-free regular formulas inside the $\langle ... \rangle$ @ operator) using the linear time algorithm $A4_{cyc}$, thus allowing to verify elaborated fairness properties on-the-fly. The presence of $\langle ... \rangle$ @ makes MCL more expressive than both $L\mu_1$ and LTL, enabling to specify the existence of complex unfair cycles in LTSs, which was previously verifiable using CADP only by means of bisimulation checking [4, 37].

Besides the operators defined in Section 2, EVALUATOR 4.0 also accepts weak modalities as in observational $L\mu$ [38]. The new tool is fully upward compatible with the versions 3.x: it accepts existing specifications written in regular $L\mu_1$, uses the same on-the-fly verification engine CÆSAR_SOLVE, offers the same diagnostic features, and keeps the same macro-definition mechanism, allowing the existing libraries of derived operators to be directly reused in MCL specifications.

Model checking of data-based fairness properties using Evaluator 4.0: We illustrate below the use of EVALUATOR 4.0 for verifying the behaviour of the SCSI-2 bus arbitration protocol [2], based on the LOTOS specification given in [16], available in the `demo_31` of CADP. The SCSI-2 protocol handles the access of devices (disks and controllers) to a bus. Devices are assigned unique numbers (priorities) in the range $[0, n-1]$. We consider a configuration containing one controller with number n_c and $n-1$ disks. The controller communicates with disk i by sending commands "*cmd i*"; after a disk receives a command, it processes it and responds to the controller with a "*rec i*" action (*reconnect* in the SCSI-2 terminology). To perform an emission or a reception, each device must get access to the bus; when several devices are requesting the bus, the device with the highest number obtains it. This priority-based arbitration raises a question about fairness: are the low priority disks always able to dialog with the controller?

It turns out that this is not the case, as it was determined experimentally by engineers at BULL [16]. The unfair behaviours of the SCSI-2 protocol are precisely captured by the following MCL formula, expressing the existence of infinite execution sequences on which disks with numbers $i < n_c$ are continuously preempted from accessing the bus by disks with higher priority:

$$[\ \mathsf{true}^*.\{cmd\ ?i{:}\mathsf{nat\ where}\ i < n_c\}\ ]$$
$$\mathsf{forall}\ j{:}\mathsf{nat\ among}\ \{i+1\ ...\ n-1\}.$$
$$(j \neq n_c) \Rightarrow \langle (\neg\{rec\ !i\})^*.\{cmd\ !j\}.(\neg\{rec\ !i\})^*.\{rec\ !j\}\rangle\ @$$

[2] See `http://www.inrialpes.fr/vasy/cadp/man/evaluator.html`

This property was impossible to express and verify precisely using the earlier versions EVALUATOR 3.x, which did not support the infinite looping operator. Using version 4.0, we checked that the above formula holds for several values of n and n_c (see the table below). The experiments were carried out on a 731 MHz, 1 GByte Pentium III machine. For each experiment, we give the size of the LTS, its generation time in seconds (for comparison with on-the-fly verification time), the size of the underlying BES, and the local resolution time. The BES contains a ν-block and a μ-block encoding the necessity modality and the infinite looping subformula, respectively. We also indicate how many times $A4_{cyc}$ was invoked for solving variables of the inner μ-block; each invocation denotes the evaluation of the $\langle ... \rangle$ @ subformula on a state reached after an appropriate "*cmd i*" action. The peak of memory usage was 182 MBytes (for $n = 5$ and $n_c = 4$). For each value of n, we observe a linear growth of the BES size and resolution time w.r.t. the value of n_c, which directly influences the effort of evaluating the formula.

cfg.		LTS		gen.	BES		res.	calls to
n	n_c	states	trans.	time	vars.	opns.	time	$A4_{cyc}$
3	0	2 060	4 630	0.73	2 061	4 631	1.30	0
	1	2 060	4 628	0.74	4 148	7 479	1.40	255
	2	2 060	4 628	0.74	4 698	8 284	1.39	255
4	0	56 169	154 752	1.71	56 170	154 753	3.52	0
	1	56 169	154 749	1.71	113 571	233 840	5.05	8 670
	2	56 169	154 749	1.71	148 444	282 024	5.87	13 005
	3	56 169	154 749	1.71	154 709	292 308	5.88	13 005
5	0	1 384 022	4 499 242	29.94	1 384 023	4 499 243	75.86	0
	1	1 384 022	4 499 238	29.78	2 710 057	6 341 224	125.74	221 085
	2	1 384 022	4 499 238	30.06	3 655 692	7 657 871	162.08	368 475
	3	1 384 022	4 499 238	30.00	4 219 664	8 446 999	182.81	442 170
	4	1 384 022	4 499 238	30.05	4 304 560	8 598 936	184.63	442 170

We successfully checked several other MCL properties on the SCSI-2 protocol, among which a safety property expressing that the difference between the number of commands received and reconnections sent by a disk i varies from 0 to 8 (the size of the buffers associated to disks):

$$\nu Y(c{:}\mathsf{nat}{:=}0).([\ \{cmd\ !i\}\ ]\ ((c < 8) \wedge Y(c+1)) \wedge [\ \{rec\ !i\}\ ]\ ((c > 0) \wedge Y(c-1)) \wedge [\ \neg(\{cmd\ !i\} \vee \{rec\ !i\})\ ]\ Y(c))$$

This property is also expressible in plain $L\mu$, but requires 9 nested maximal fixed point operators (one for each value of the counter c) and 27 box modalities.

5 Conclusion and Future Work

The specification of temporal properties in $L\mu$ is a difficult task that requires a significant training and experience. By proposing MCL, a user-friendly extension of $L\mu$, we attempted to facilitate this task for branching-time, action-based properties interpreted on LTSs; our effort goes in the same direction as existing enhancements of state-based temporal logics [3, 5, 6, 23]. Our model checking

method, based on reformulating the problem as a BES resolution, provides a natural evaluation engine for parameterized fixed points on finite LTSs; infinite-state systems could be handled using symbolic resolution of PBESs [18]. The restriction to the alternation-free MCL fragment, motivated by efficiency, is compensated by the ability of the infinite looping operator to handle fairness properties. The local BES resolution algorithm $A4_{cyc}$ that we proposed for evaluating this operator yields a linear-time on-the-fly model checking procedure for PDL-Δ, despite its embedding in $L\mu_2$ [13].

We plan to continue our work along several directions. Firstly, a tighter coupling is needed between EVALUATOR 4.0 and the data types and functions of the program under verification: this can be done by appropriately extending the LTS exploration API defined by OPEN/CÆSAR with data manipulation features. Secondly, MCL can be enhanced with the operators of other logics, such as (action-based) LTL, which can be translated using the infinite looping operator. Finally, a distributed version of EVALUATOR 4.0 can be obtained by coupling it with distributed BES resolution algorithms [25].

References

1. H. R. Andersen. Model Checking and Boolean Graphs. *TCS*, 126(1):3–30, 1994.
2. ANSI. Small Computer System Interface-2. Standard X3.131-1994.
3. R. Armoni, L. Fix, A. Flaisher, *et al.* The ForSpec Temporal Logic: A New Temporal Property-Specification Language. *TACAS'02*, LNCS vol. 2280, pp. 296–211.
4. T. Arts, C. Benac Earle, and J. Derrick. Development of a Verified Erlang Program for Resource Locking. *STTT*, 5(2–2):205–220, 2004.
5. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. *VMCAI'04*, LNCS vol. 2937, pp. 44–57.
6. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. *CAV'01*, LNCS vol. 2102, pp. 363–367.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
8. R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. *FMSD*, 2(2):121–147, 1993.
9. M. Dam. Model Checking Mobile Processes (Full version). Research Report RR 94:1, Swedish Institute of Computer Science, Kista, Sweden, 1994.
10. R. De Nicola and F. W. Vaandrager. Action versus State Based Logics for Transition Systems. *Semantics of Concurrency*, LNCS vol. 469, pp. 407–419, 1990.
11. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. *ICSE'99*, pp. 411–420.
12. E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *JACM*, 33(1):151–178, 1986.
13. E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. *LICS'86*, pp. 267–278, 1986.
14. M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *JCSS*, 18(2):194–211, 1979.
15. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. *TACAS'98*, LNCS vol. 1384, pp. 68–84.
16. H. Garavel and H. Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. *FME'02*, LNCS vol. 2391, pp. 410–429.

17. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. *CAV'07*, LNCS vol. 4590, pp. 158–163.
18. J. F. Groote and T. A. C. Willemse. Parameterised Boolean Equation Systems. *TCS*, 343:332–369, 2005.
19. J. F. Groote and R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. *AMAST'98*, LNCS vol. 1548, pp. 74–90.
20. J. Y. Halpern and J. H. Reif. The Propositional Dynamic Logic of Deterministic, Wellstructured Programs. *TCS*, 27(1–2):127–165, 1983.
21. K. Hamaguchi, H. Hiraishi, and S. Yajima. Branching Time Regular Temporal Logic for Model Checking with Linear Time Complexity. *CAV'90*, LNCS vol. 531.
22. G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
23. IEEE. PSL: Property Specification Language. Std. P1850, IEEE, 2004.
24. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Int. Std. 8807, ISO — OSI, Genève, 1989.
25. C. Joubert and R. Mateescu. Distributed On-the-Fly Model Checking and Test Case Generation. *SPIN'06*, LNCS vol. 3925, pp. 126–145.
26. D. Kozen. Results on the Propositional μ-calculus. *TCS*, 27:333–354, 1983.
27. K. G. Larsen. Proof Systems for Hennessy-Milner logic with Recursion. *CAAP'88*, LNCS vol. 299, pp. 215–230.
28. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. VERSAL 8, Bertz Verlag, Berlin, 1997.
29. R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. *VMCAI'98*. University Ca'Foscari of Venice, 1998.
30. R. Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. *TACAS'00*, LNCS vol. 1785, pp. 251–265.
31. R. Mateescu. CÆSAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *STTT*, 8(1):37–56, 2006.
32. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *SCP*, 46(3):255–281, 2003.
33. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
34. A. Pnueli. A Temporal Logic of Concurrent Programs. *TCS*, 13:45–60, 1981.
35. J-P. Queille and J. Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.
36. J. Rathke and M. Hennessy. Local Model Checking for a Value-Based Modal μ-calculus. Report 5/96, Univ. of Sussex, 1996.
37. G. Salaün, W. Serwe, Y. Thonnart, and P. Vivet. Formal Verification of CHP Specifications with CADP — Illustration on an Asynchronous Network-on-Chip. *ASYNC'07*, IEEE, pp. 73–82.
38. C. Stirling. *Modal and Temporal Properties of Processes*. Springer Verlag, 2001.
39. R. Streett. Propositional Dynamic Logic of Looping and Converse. *Information and Control*, 54:121–141, 1982.
40. R. E. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM J. of Computing*, 1(2):146–160, 1972.
41. W. Thomas. Computation Tree Logic and Regular ω-languages. LNCS vol. 354.
42. B. Vergauwen and J. Lewi. A Linear Algorithm for Solving Fixed-Point Equations on Transition Systems. *CAAP'92*, LNCS vol. 581, pp. 322–341.
43. P. Wolper. A Translation from Full Branching Time Temporal Logic to One Letter Propositional Dynamic Logic with Looping. Unpublished manuscript, 1982.
44. P. Wolper. Temporal Logic Can Be More Expressive. *Information and Control*, 56(1/2):72–99, 1983.