Algorithms **2014**, *7*, 339-362; doi:10.3390/a7030339

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

Model Checking Properties on Reduced Trace Systems

Antonella Santone [1,*] and Gigliola Vaglini [2,*]

[1] Dipartimento di Ingegneria, Università del Sannio, Benevento 82100, Italy

[2] Dipartimento di Ingegneria della Informazione, Università di Pisa, Pisa 56122, Italy

* Authors to whom correspondence should be addressed; E-Mails: santone@unisannio.it (A.S.); g.vaglini@iet.unipi.it (G.V.); Tel.: +39-0824-305552 (A.S); +39-050-2217528 (G.V.).

Received: 7 February 2014; in revised form: 16 May 2014 / Accepted: 24 June 2014 / Published: 8 July 2014

Abstract: Temporal logic has become a well-established method for specifying the behavior of distributed systems. In this paper, we interpret a temporal logic over a partial order model that is a trace system. The satisfaction of the formulae is directly defined on traces on the basis of rewriting rules; so, the graph representation of the system can be completely avoided; moreover, a method is presented that keeps the trace system finite, also in the presence of infinite computations. To further reduce the complexity of model checking temporal logic formulae, an abstraction technique is applied to trace systems.

Keywords: trace systems; model checking; linear and branching time temporal logic; state explosion

---

1. Introduction and Motivation

Linear time [1] and branching time [2] temporal logics are used for specifying and verifying concurrent and distributed systems: partial order models (trace systems are an example) are mostly used to give semantics to linear time logics, while interleaving models (such as transition systems) are widely used for branching time logics. To express properties inherent in concurrency, *i.e.*, properties distinguishing concurrency from nondeterminism, a partial order interpretation for the logic fits better. This interpretation allows also a good definition of fairness properties as, for example, inevitability under the fairness assumption [3,4]: "in all computations the event *a* eventually occurs".

Model checking is one of the main methods for the automated verification of concurrent systems [5]; it consists in checking whether a structure representing the system is a model for a logic formula. Model

checking very large concurrent systems may cause the so-called "state explosion problem" and lead to a too big number of states of the structure. A variety of methods for reducing the state explosion problem have been developed [6–10]; in the context of branching time logics, in [11,12], the authors and others proposed a logic, called selective mu-calculus, equi-expressive to mu-calculus [13], but, such that each formula directly characterizes an abstraction of the system that maintains the truth value of the formula itself. Different action logics, such as [14], whose operators could be also used in a linear fashion to concisely express fairness properties, are not suitable to individuate system abstractions preserving the truth values of formulae. A further problem is the model checking of infinite representation of systems: for example, in trace systems, recursive behaviors are usually represented by means of an infinite set of finite traces (see [15], solutions for branching time logic are in [16]). Finally, it is known that, while a lot of interesting correctness properties, such as mutual exclusion and the absence of starvation, can be elegantly expressed by linear time formulae, the model checking of a linear time logic and that of a branching time logic [5,17,18] have different complexity. For example, given a transition system of size n and an alternation-free temporal logic formula of size m, model checking algorithms for the branching time logic (CTL)run in time $\mathcal{O}(nm)$, while those for the linear time logics (LTL) run in time $\mathcal{O}(n2^m)$. This result holds also in the case of generalized model checking [19].

In this work, we give a non-interleaving interpretation of selective mu-calculus formulae using the simplest and best known partial order model for computations, that is Mazurkiewicz's trace system [3,15,20]. This model allows a compact representation of the system computations using only an element, called trace, to represent an equivalence class of sequences of events with respect to a dependence relation. A similar approach has been carried on with the logic CTL in [21]. More precisely, the author defines an extension of CTL by past modalities, called CTL_P, and interpreted over Mazurkiewicz's trace systems. The author's aim is to obtain the model checking of properties described in this logic with a linear complexity, as is that of CTL on transition systems: on the contrary, he proved that model checking for CTL_P on traces is NP-hard, even if past modalities cannot be nested. Differently from [21], the aim of this article is to check the satisfaction of our formulae directly on traces, *i.e.*, we suppose using a sequential memory representation for traces and no graph representation of the system, but only a representation of dependencies. Moreover, we employ a trace abstraction, induced by the selective formulae, and we simplify the traces by discarding, at each verification step, the events that are no more of interest for the successive verification. Furthermore, to avoid the management of infinite sets of traces, we use partial traces containing holes to give the semantics of recursive behaviors: the holes are expanded step-by-step, until the verification of the formula can be decided, and so, we have always to manage a finite set of traces. The checking method can be easily implemented by rewriting functions that transform each trace with a polynomial complexity in its dimension.

In Section 2, concurrent systems are defined by a simple event-based specification language, taken as an example and whose semantics is a trace system. In Section 3, the syntax of the selective mu-calculus is recalled, and the satisfaction of the formulae is defined on the corresponding trace system. In the successive section, this is shown as each trace can be abstracted with respect to a particular formula, but maintaining its truth value. The last section contains conclusions and comparisons of the presented approach with some related works.

2. Event Language

This section presents a very simple, event-based language that is not a real language, but an exercise one; nevertheless, it contains the basic features to represent behaviors of concurrent systems. The language is actually very similar to some of the most common process algebras. The semantics of the language is given in terms of trace systems.

2.1. Syntax of Expressions

Expressions are obtained composing a finite set $A = \{a, b, \ldots\}$ of symbols, called the alphabet, by means of a set of operators. Each expression represents a possible behavior of the system, while the occurrence of a symbol in an expression represents the occurrence of an event of the system. The syntax to build up expressions is the following:

$$e \quad ::= \quad nil \mid a \mid e.e \mid e + e \mid e \| e \mid rec(e)$$

where a ranges over A. The language allows the definition of:

- the empty expression (the operator nil);
- the concatenation of two expressions (the operator "."); for example, $e_1.e_2$, with $e_1 = a$ and $e_2 = b$, is the expression $a.b$;
- the choice between two expressions (the operator "+"); for example, $e_1 + e_2$, with $e_1 = a.c$ and $e_2 = b$, is the expression $a.c + b$;
- the parallel composition of two expressions (operator "$\|$"), where the events in each expression can occur independently, except the events with the same name that cause the synchronization of the two concurrent expressions; for example, $e_1 \| e_2$, with $e_1 = a.c$ and $e_2 = b$, is the expression $a.c \| b$;
- the unbounded iteration of an expression (the operator "rec").

We require that the following rules hold for the expressions:

$$1.\ e.nil = e \quad 2.\ nil \| e = e \quad 3.\ rec(nil) = nil \quad 4.\ e \| e = e \quad 5.\ e + e = e$$

while $e + nil = e$ does not hold; the trace semantics of the language is given by the Definition 2 in the next section, and it guarantees the rules.

For each alphabet A, A^* is the set of all finite sequences (strings) of symbols in A; for each string σ, $alph(\sigma)$ (the alphabet of σ) is the set of all symbols occurring in σ; this definition can be easily extended over expressions. We denote the set of all expressions by $\mathcal{E}$. In the following section, the semantics of the language is formally given.

2.2. Trace Semantics of Expressions

In this section, we define the trace semantics of expressions of the language.

The first step is the definition of the notion of dependence between events; the relation we use is taken from Mazurkiewicz's trace theory [3,15].

Dependence. A dependence relation (dependence for short) over a finite alphabet A is any reflexive, symmetric relation $D \subseteq A \times A$. A dependence D over the alphabet A determines the symmetric and irreflexive relation $I_D \subseteq A \times A$, called the independence relation determined by D and defined as $I_D = A \times A - D$.

The ordered triple $(A^*, ., \epsilon)$, where ϵ is the empty string and . is the concatenation operation on strings, is the standard string monoid over A. Usually, the sign . for concatenation is omitted. Let A be an alphabet and $\sigma \in A^*$; then for any alphabet B we denote $\Pi_B(\sigma)$ the string projection of σ onto B, *i.e.*, a string over $A \cap B$ obtained from σ deleting all symbols not belonging to B. If $A \cap B = \emptyset$, $\Pi_B(\sigma) = \epsilon$, for any σ. A concurrent alphabet is any ordered pair $\Sigma = (A, D)$, where A is a finite set of symbols, called also the alphabet of Σ, and D is a dependence over A. Given $\Sigma = (A, D)$, the trace equivalence for Σ is the least congruence $\equiv_\Sigma$ in the string monoid over A, such that for all a, b:

$$(a, b) \in I_D \Rightarrow ab \equiv_\Sigma ba$$

In other words, it holds that $\sigma \equiv_\Sigma \sigma'$ if there is a finite sequence of strings $\sigma_0, \sigma_1, \ldots, \sigma_n,\ n \geq 0$, such that $\sigma_0 = \sigma,\ \sigma_n = \sigma'$, and for each i, $1 \leq i \leq n$, $\sigma_{i-1} = \delta_1 ab \delta_2$, $\sigma_i = \delta_1 ba \delta_2$, for some $(a, b) \in I_D$ and $\delta_1, \delta_2 \in A^*$. Equivalence classes of $\equiv_\Sigma$ are called traces over Σ; given a string $\sigma \in A^*$, $[\sigma]_\Sigma$ is the equivalence class of σ over Σ. When clear from the context, we omit Σ. The set $\Theta(\Sigma) = [A^*]_\Sigma$ is the set of all traces over Σ. The concatenation of traces, denoted by $[\sigma_1][\sigma_2]$, is defined as $[\sigma_1\sigma_2]$. For any alphabet B, it holds that $\Pi_B([\sigma]) = [\Pi_B(\sigma)]$ and $alph([\sigma]) = alph(\sigma)$.

Dependence Closure. Given two dependencies, D_1 and D_2, defined on the alphabets A_1 and A_2, respectively, we call dependence closure the derived dependence on the alphabet $A_1 \cup A_2$:

$$D_1 D_2^\circ = D_1 \cup D_2 \cup \{(a_1, a_2), (a_2, a_1) \mid a_1 \in A_1, a_2 \in A_2\}$$

Trace System. A trace system TS is any ordered pair (Σ, T), where $\Sigma = (A, D)$ is a concurrent alphabet and $T \subseteq \Theta(\Sigma)$ is a trace language over Σ.

To manage only finite sets of traces also in the presence of recursive behaviors, we extend the alphabet A by a set of special symbols (called holes): a hole represents the fact that the trace is incomplete and can be expanded; we also call this type of trace a partial trace. Each hole has the form $\lfloor\ \rfloor_x$, where x is the name of a trace language; in fact, any trace belonging to the language x can be used to fill the hole giving rise to another partial trace.

Some operations can be performed on trace systems.

- Given the trace system $TS = ((A, D), T)$,
  - its unbounded iteration is the system:

$$TS_1 = (((A, D), T))^* = ((A \cup \{\lfloor\ \rfloor_T\}, D \cup D'), \{[\epsilon], \lfloor\ \rfloor_T\})$$

where:

$$D' = \{(\lfloor\ \rfloor_T, \lfloor\ \rfloor_T)\} \cup \{(a, \lfloor\ \rfloor_T), (\lfloor\ \rfloor_T, a) \mid a \in A\}$$

The following example gives a first hint of the effect of the unbounded iteration of trace systems.

Example 1. Consider the trace system $TS_0 = ((A_0, D_0), T_0)$, with:
$A_0 = \{a\}$, $D_0 = \{(a, a)\}$ and $T_0 = \{[a]\}$

$$(TS_0)^* = ((\{a\} \cup \{\lfloor\ \rfloor_{T_0}\}, \{(a, a), (\lfloor\ \rfloor_{T_0}, \lfloor\ \rfloor_{T_0}), (a, \lfloor\ \rfloor_{T_0}), (\lfloor\ \rfloor_{T_0}, a)\}), \{[\epsilon], \lfloor\ \rfloor_{T_0}\})$$

The actual traces of $(TS_0)^*$ are all of the partial traces that can be obtained by filling its hole by means of the traces of T_0: some examples are $[a \lfloor \ \rfloor_{T_0}], [aa \lfloor \ \rfloor_{T_0}], [aaa \lfloor \ \rfloor_{T_0}]$; Definition 1 will show the formal way in which more complete traces can be obtained. Other operations on trace systems are the following ones.

- Let $TS_1 = (\Sigma_1, T_1)$ and $TS_2 = (\Sigma_2, T_2)$, with $\Sigma_1 = (A_1, D_1)$ and $\Sigma_2 = (A_2, D_2)$, be two trace systems.
  - Their concatenation is the system:

$$TS_1.TS_2 = (\Sigma', T')$$

$$\text{where: } \Sigma' = (A_1 \cup A_2, D_1 D_2^{\circ}), \text{ and } T' = \{\tau_1\tau_2 \mid \tau_1 \in T_1,\ \tau_2 \in T_2\}$$

  - their nondeterministic composition is the system:

$$TS_1 + TS_2 = (\Sigma', T')$$

$$\text{where } \Sigma' = (A_1 \cup A_2, D_1 \cup D_2), \text{ and}$$

$$T' = \{\tau \mid \tau \in T_1 \vee \tau \in T_2\}$$

  - their parallel composition is the system:

$$TS_1 \| TS_2 = (\Sigma', T')$$

$$\text{where } \Sigma' = (A_1 \cup A_2, D_1 \cup D_2), \text{ and}$$

$$T' = \{\Pi_{A_1}(\tau) \in T_1 \wedge \Pi_{A_2}(\tau) \in T_2\}$$

Example 2. *Consider the following trace systems :*
$TS_0 = ((A_0, D_0), T_0)$,
$TS_1 = ((A_1, D_1), T_1)$, *and*
$TS_2 = ((A_2, D_2), T_2)$, *with*
$A_0 = \{a\}$, $D_0 = \{(a, a)\}$ *and* $T_0 = \{[a]\}$;
$A_1 = \{a, b\}$, $D_1 = \{(a, a), (b, b), (a, b), (b, a)\}$ *and* $T_1 = \{[ab]\}$;
$A_2 = \{b, c\}$, $D_2 = \{(b, b), (c, c), (b, c), (c, b)\}$ *and* $T_2 = \{[bc]\}$.

$TS_0.TS_2$ *is the trace system* $TS_3 = ((A_3, D_3), T_3)$ *with*
$A_3 = \{a, b, c\}$, $D_3 = \{(a, a), (b, b), (c, c), (a, b), (b, a), (a, c), (c, a), (b, c), (c, b)\}$ *and* $T_3 = \{[abc]\}$.

$TS_0 + TS_2$ *is the trace system* $TS_6 = ((A_6, D_6), T_6)$ *with*
$A_6 = \{a, b, c\}$, $D_6 = \{(a, a), (b, b), (c, c), (b, c), (c, b)\}$ *and* $T_6 = \{[a], [bc]\}$.

$TS_1 \| TS_2$ *is the trace system* $TS_4 = ((A_4, D_4), T_4)$ *with*
$A_4 = \{a, b, c\}$, $D_4 = \{(a, a), (b, b), (c, c), (a, b), (b, a), (b, c), (c, b)\}$ *and* $T_4 = \{[abc]\}$.

$TS_0 \| TS_2$ *is the trace system* $TS_5 = ((A_5, D_5), T_5)$ *with*
$A_5 = \{a, b, c\}$, $D_5 = \{(a, a), (b, b), (c, c), (b, c), (c, b)\}$ *and* $T_5 = \{[abc]\}$.

The expansion of the partial traces is obtained through the following definition: a completion step is performed by prefixing each hole $\lfloor\ \rfloor_T$ in the partial trace by means of an element of the trace language T; the same occurs for the holes possibly contained in T. Each expansion of the partial trace maintains the capability of a further expansion for each hole.

Definition 1 (one-unfolding)**.**
Consider $TS = ((A, D), T)$ and $\sigma \in A^*$, all the possible one-unfoldings of σ, denoted by $\widehat{\sigma}$, correspond to the set $\mathcal{U}(\sigma, S)$, obtained as follows from an initial value of $S = \emptyset$:

$$\mathcal{U}(\sigma, S) = \begin{cases} \sigma & \text{if } \sigma \text{ contains no hole or} \\ & \forall \lfloor\ \rfloor_x \in \sigma,\ x \in S \\ \mathcal{U}(\sigma\left[^{\sigma_1 \lfloor\ \rfloor_x}/_{\lfloor\ \rfloor_x}\right], S \cup \{x\}) & \\ \cdots & \text{if } \lfloor\ \rfloor_x \in \sigma,\ x \notin S, \forall i \in [1..n], \sigma_i \in x \\ \mathcal{U}(\sigma\left[^{\sigma_n \lfloor\ \rfloor_x}/_{\lfloor\ \rfloor_x}\right], S \cup \{x\}) & \end{cases}$$

Following the previous definition, if $x = \{[dg], [df \lfloor\ \rfloor_y]\}$ and $y = \{[dc]\}$; for example, the trace $[abc \lfloor\ \rfloor_x d]$ may be transformed by filling the hole $\lfloor\ \rfloor_x$ with the first trace in the language x, so obtaining the partial trace $[abcdg \lfloor\ \rfloor_x d]$. When using the second partial trace in x, we obtain the partial trace $[abcdf dc \lfloor\ \rfloor_y \lfloor\ \rfloor_x d]$, since also the hole $\lfloor\ \rfloor_y$ must be filled one time. The one-unfolding procedure of a string always terminates after each hole in the initial trace (and each hole in the traces used to fill it) has been filled once.

The semantics of an expression is the trace system built on the basis of the syntactic structure of each expression.

Definition 2 (Semantics)**.** *Given the expression e, its semantics is the trace system $TS(e) = ((A, D), T)$ built as follows:*

$$\begin{aligned} TS(nil) &= ((\emptyset, \emptyset), \{[\epsilon]\}) \\ TS(a) &= ((\{a\}, \{(a, a)\}), \{[a]\}) \\ TS(e_1.e_2) &= TS(e_1).TS(e_2) \\ TS(e_1 + e_2) &= TS(e_1) + TS(e_2) \\ TS(e_1 \| e_2) &= TS(e_1) \| TS(e_2) \\ TS(rec(e)) &= (TS(e))^* \end{aligned}$$

We remark that the rule $e + nil = e$ does not hold; in fact, $TS(e + nil)$ always contains the empty trace, while $TS(e)$ may not.

The following example clarifies the semantics of the rec operator.

Example 3. *Consider the expression:*

$$e = rec(a.rec(d))$$

$TS(rec(a.rec(d))) = (TS_0(a.rec(d)))^*$
$TS_0(a.rec(d))) = TS_1(a).TS_2(rec(d))$
$TS_2(rec(d)) = (TS_3(d))^*$
If $TS = ((A, D), T)$, *we have:*

$$\begin{aligned} A &= A_0 \cup \{\lfloor\ \rfloor_{T_0}\} \\ D &= D_0 \cup \{(\lfloor\ \rfloor_{T_0}, \lfloor\ \rfloor_{T_0})\} \cup \{(a, \lfloor\ \rfloor_{T_0}), (\lfloor\ \rfloor_{T_0}, a) \mid \forall a \in A_0\} \\ T &= \{[\epsilon], \lfloor\ \rfloor_{T_0}\} \end{aligned}$$

where, for each $0 \leq i \leq 3$, $TS_i = ((A_i, D_i), T_i)$ *is defined as follows:*

$$\begin{aligned} A_0 &= \{a, d, \lfloor\ \rfloor_{T_3}\} \\ A_1 &= \{a\} \\ A_2 &= A_3 \cup \{\lfloor\ \rfloor_{T_3}\} \\ A_3 &= \{d\} \\ D_0 &= D_1 D_2^\circ = \{(a,a),\ (d,d),\ (\lfloor\ \rfloor_{T_0}, \lfloor\ \rfloor_{T_0}),\ (\lfloor\ \rfloor_{T_3}, \lfloor\ \rfloor_{T_3}),\ (a,d),\ (d,a), \\ &\qquad (a, \lfloor\ \rfloor_{T_3}),\ (\lfloor\ \rfloor_{T_3}, a), (a, \lfloor\ \rfloor_{T_0}),\ (\lfloor\ \rfloor_{T_0}, a),\ (d, \lfloor\ \rfloor_{T_0}),\ (\lfloor\ \rfloor_{T_0}, d), \\ &\qquad (\lfloor\ \rfloor_{T_3}, \lfloor\ \rfloor_{T_0}),\ (\lfloor\ \rfloor_{T_0}, \lfloor\ \rfloor_{T_3}), (d, \lfloor\ \rfloor_{T_3}),\ (\lfloor\ \rfloor_{T_3}, d)\} \\ D_1 &= \{(a,a)\} \\ D_2 &= D_3 \cup \{(\lfloor\ \rfloor_{T_3}, \lfloor\ \rfloor_{T_3}),\ (\lfloor\ \rfloor_{T_3}, d), (d, \lfloor\ \rfloor_{T_3}),\ (\lfloor\ \rfloor_{T_3}, d))\} \\ D_3 &= \{(d,d)\} \\ T_0 &= \{[a], [a \lfloor\ \rfloor_{T_3}]\} \\ T_1 &= \{[a]\} \\ T_2 &= \{[\epsilon], \lfloor\ \rfloor_{T_3}\} \\ T_3 &= \{[d]\} \end{aligned}$$

The strings below are the one-unfoldings of $\lfloor\ \rfloor_{T_0}$*:*

$$a \lfloor\ \rfloor_{T_0}, ad \lfloor\ \rfloor_{T_3} \lfloor\ \rfloor_{T_0}$$

3. Selective Mu-Calculus

The selective mu-calculus is a temporal logic proposed by the authors in [11,12] and interpreted on transition systems, as a branching time logic. That calculus has the characteristic that the actions relevant for checking a formula are the ones explicitly mentioned. We propose here a different interpretation that takes into account linear time; the following sub-section recalls the syntax of the calculus (called LTSC, for short) and the satisfaction of LTSC formulae on trace systems.

3.1. The Syntax of the Calculus

Here, slight simplifications are made on the syntax of the selective mu-calculus to avoid useless details. Consider the set A of events. The events a, b range over A, and $S \subseteq A$ is a set of events with cardinality less than or equal to one. Moreover, Z belongs to a set of variable names. The calculus has the following syntax:

$$\varphi \quad ::= \quad \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [a]_S\varphi \mid \langle a\rangle_S\varphi \mid \nu Z.\varphi \mid \mu Z.\varphi$$

A fixed point formula has the form $\mu Z.\varphi$ ($\nu Z.\varphi$), where the fixed point operator μZ (νZ) binds free occurrences of Z in φ. An occurrence of Z is free if it is within the scope of no fixed point operator. A formula is closed if it contains no free variables. The formula $\mu Z.\varphi$ is the least fixed point of the recursive equation $Z = \varphi$, while $\nu Z.\varphi$ is the greatest one. In the following, we consider only closed formulae and alternation-free mu-calculus formulae [22].

However, note that the syntax of the modal operators can be easily extended as follows (and so their meaning) to manage a set of events without affecting the remainder of the paper.

$$[\{\alpha_1, \ldots, \alpha_n\}]_{\{\beta_1,\ldots,\beta_m\}}\varphi =$$
$$\Big([\alpha_1]_{\{\beta_1\}}\varphi \vee \cdots \vee [\alpha_1]_{\{\beta_m\}}\varphi\Big) \wedge \cdots \wedge \Big([\alpha_n]_{\{\beta_1\}}\varphi \vee \cdots \vee [\alpha_n]_{\{\beta_m\}}\varphi\Big)$$
$$\langle\{\alpha_1, \ldots, \alpha_n\}\rangle_{\{\beta_1,\ldots,\beta_m\}}\varphi =$$
$$\Big(\langle\alpha_1\rangle_{\{\beta_1\}}\varphi \wedge \cdots \wedge \langle\alpha_1\rangle_{\{\beta_m\}}\varphi\Big) \vee \cdots \vee \Big(\langle\alpha_n\rangle_{\{\beta_1\}}\varphi \wedge \cdots \wedge \langle\alpha_n\rangle_{\{\beta_m\}}\varphi\Big)$$

Example 4. *Some examples of formulae are shown in the following.*

$\varphi_1 = \nu Z.\langle a\rangle_\emptyset Z$*: "there exists a run in which the event* a*, preceded by any event, can always occur".*

$\varphi_2 = [c]_{\{a\}}\langle a\rangle_\emptyset\mathtt{tt}$*: "in any run where an event* c*, not preceded by the event* a*, occurs, the event* a *must always follow.*

3.2. The Satisfaction of the Formulae on Trace Systems

To define the formula satisfaction on trace systems, we will build a particular trace presentation; the behavior of the operators that produce this presentation is based on a rewriting rule, which transforms a trace into an equivalent one.

Definition 3 (Rewriting rule)**.** *Consider* $\Sigma = (A, D)$ *and* $\gamma ba\gamma' \in A^*$.

$$\gamma ba\gamma' \mapsto \gamma ab\gamma' \quad \textit{if } (b, a) \in I_D$$

The rule transforms a string into an equivalent one, since:

$$(b, a) \in I_D \Rightarrow \gamma ba\gamma' \equiv_\Sigma \gamma ab\gamma'$$

The following function $\mathcal{D}$ uses the auxiliary function $\mathcal{M}$, shown in Table 1. The function $\mathcal{M}$, given a string β, marks all the events in β. If A is an alphabet, we write A^- for the set $\{a \mid a \in A \wedge a \neq \lfloor \rfloor_x, \forall \lfloor \rfloor_x \in A\}$. Intuitively, A^- is the set of all of the symbols of the alphabet A, except for the holes.

Table 1. Marking function.

Let A be an alphabet: consider $a \in A$ and $\beta \in A^*$.

$$\begin{aligned} \mathcal{M}(a.\beta) &= \bar{a}.\mathcal{M}(\beta) \\ \mathcal{M}(\epsilon) &= \epsilon \end{aligned}$$

Definition 4. *Consider* $\Sigma = (A, D)$, $\sigma \in A^*$, $a \in A^-$, *and* $S \subseteq A^-$.

$\mathcal{D}_{a,S}(\sigma) =$

$$\begin{cases} [\mathcal{M}(\delta_2).\delta_3] & (1)\ \textit{if}\ \exists \sigma_1 a \sigma_2 \equiv_\Sigma \sigma,\ \Pi_{\{a\}\cup S}(\sigma_1) = \epsilon;\ \textit{and} \\ & (2)\ \textit{if}\ \exists \delta_1\delta_2\ a\ \delta_3 \equiv_\Sigma \delta_1\ a\ \delta_2\delta_3 \equiv_\Sigma \sigma_1\ a\ \sigma_2,\ \textit{such that} \\ & \quad (a)\ \Pi_{\{a\}\cup S}(\delta_1\delta_2) = \epsilon;\ \textit{and} \\ & \quad (\mathbf{b})\ \textit{no event in}\ \delta_1\ \textit{can move forward (by means of the rewriting rule),} \\ & \quad \textit{to pass over}\ a;\ \textit{and} \\ & \quad (c)\ \textit{no event in}\ \delta_3\ \textit{can move backward to pass over}\ a \\ & \quad \textit{(by means of the rewriting rule) apart the event in S.} \\ null & \textit{otherwise} \end{cases}$$

$\mathcal{D}_{a,S}(\sigma)$ manipulates traces exploiting a simple algorithm that moves events according to the rewriting rule. The result of $\mathcal{D}_{a,S}(\sigma)$ is a trace without the events occurring before a in any run (the events in δ_1 do not matter for the successive verification), but that includes the events occurring in some run after a and in some run before a (they are the marked events in δ_2, with the mark remembering that they do not necessarily occur where they are); it includes also (they are the events in δ_3) the events that occur after a in any run and, if present, the event in S (such an event is required to occur after a in all runs of interest; this fact is guaranteed by the constraint expressed in Point a) of Definition 4). The complexity of the trace manipulations is polynomial in the number of elements of the trace itself in the worst case, when at each step, all of the events are moved up and down to verify if they can occur before and/or after a; linear in the better case, when only the reading of all of the events of the trace is required, since the rewriting rule cannot ever be applied. The following example shows, in more detail, the effect of $\mathcal{M}$.

Example 5. *Consider the two concurrent alphabets:*
$\Sigma_1 = (\{a, b, c\},\ \{(a, a),\ (b, b),\ (c, c),\ (a, b),\ (b, a)\})$ *and:*
$\Sigma_2 = (\{a, b, c\},\ \{(a, a),\ (b, b),\ (c, c),\ (b, c),\ (c, b)\})$.

- *If* $S = \emptyset$*:*

  $\mathcal{D}_{a,S}(cab) = [\bar{c}b]$, *for* Σ_1, *and*

  $\mathcal{D}_{a,S}(cab) = [\bar{c}\bar{b}]$, *for* Σ_2, *while*

  $\mathcal{D}_{b,S}(cab) = [\bar{c}]$, *for* Σ_1, *and*

  $\mathcal{D}_{b,S}(cab) = [\bar{a}]$, *for* Σ_2

- *If* $S = \{b\}$

  $\mathcal{D}_{a,S}(cab) = [\bar{c}b]$ *for both* Σ_1 *and* Σ_2, *while*

  $\mathcal{D}_{b,S}(cab) = [\bar{c}]$, *for* Σ_1, *and*

  $\mathcal{D}_{b,S}(cab) = [\bar{a}]$, *for* Σ_2

The formal satisfaction of a formula ψ by a trace $[\sigma]$ is defined as follows. Note that we consider the event set $\widetilde{X} = X \cup \bar{X}$, where $\bar{X} = \{\bar{x} | x \in X\}$ as the alphabet for strings, since it is possible to obtain strings containing marked events. In Table 2, auxiliary cleaning functions are shown that either eliminate the marks from the events of a string ($\mathcal{C}l^2$) before the next verification step is performed or eliminate the marked events at all ($\mathcal{C}l^1$), as shown in the following Example 6. In fact, the marked events must not be considered when checking for the satisfaction of a formula $\langle a \rangle_S \psi$.

Table 2. Cleaning functions.

Let A be an alphabet; consider $a, b \in A$ and $\beta \in A^*$.

$$
\begin{array}{lcl}
\mathcal{C}l^1(b, a.\beta) & = & a.\mathcal{C}l^1(b, \beta) \\
\mathcal{C}l^1(b, \bar{a}.\beta) & = & \text{if } (a = b) \text{ then } \mathcal{C}l^1(b, \beta) \text{ else } a.\mathcal{C}l^1(b, \beta) \\
\mathcal{C}l^1(b, \epsilon) & = & \epsilon \\
\mathcal{C}l^2(a.\beta) & = & a.\mathcal{C}l^2(\beta) \\
\mathcal{C}l^2(\bar{a}.\beta) & = & a.\mathcal{C}l^2(\beta) \\
\mathcal{C}l^2(\epsilon) & = & \epsilon
\end{array}
$$

Definition 5. *Consider* $\Sigma = (A, D)$, $a \in A^-$, $S \subseteq A^-$, $\sigma \in \widetilde{A}^*$; *moreover,* $\widehat{\sigma}$ *is any one-unfolding of* σ.

$$
\begin{array}{lll}
[\sigma] \models \mathtt{tt} & & \\
[\sigma] \not\models \mathtt{ff} & & \\
[\sigma] \models \psi_1 \vee \psi_2 & \textit{iff} & [\sigma] \models \psi_1 \textit{ or } [\sigma] \models \psi_2 \\
[\sigma] \models \psi_1 \wedge \psi_2 & \textit{iff} & [\sigma] \models \psi_1 \textit{ and } [\sigma] \models \psi_2 \\
[\sigma] \models [a]_S\, \psi & \textit{iff} & \forall \sigma' \in \widehat{\sigma},\ \textit{such that } \mathcal{D}_{a,S}(\mathcal{C}l^2(\sigma')) \neq null,\ \mathcal{D}_{a,S}(\mathcal{C}l^2(\sigma')) \models \psi \\
[\sigma] \models \langle a \rangle_S\, \psi & \textit{iff} & \exists \sigma' \in \widehat{\sigma},\ \textit{such that } \mathcal{D}_{a,S}(\mathcal{C}l^1(a, \sigma')) \neq null \\
 & & \textit{and } \mathcal{D}_{a,S}(\mathcal{C}l^1(a, \sigma')) \models \psi \\
[\sigma] \models \nu Z.\psi & \textit{iff} & [\sigma] \models \nu Z^n.\psi \textit{ for all natural numbers } n \\
[\sigma] \models \mu Z.\psi & \textit{iff} & [\sigma] \models \mu Z^n.\psi \textit{ for some natural number } n
\end{array}
$$

where, for each n, $\nu Z^n.\varphi$ *and* $\mu Z^n.\varphi$ *are defined as:*

$$
\begin{array}{ll}
\nu Z^0.\varphi = \mathtt{tt} & \mu Z^0.\varphi = \mathtt{ff} \\
\nu Z^{n+1}.\varphi = \varphi[\nu Z^n.\varphi/Z] & \mu Z^{n+1}.\varphi = \varphi[\mu Z^n.\varphi/Z]
\end{array}
$$

and the notation $\varphi[\psi/Z]$ *indicates the substitution of* ψ *for every free occurrence of the variable* Z *in* φ.

Example 6. *Reconsider the previous Example 5 and the formula:*

$$\varphi_3 = [a]_\emptyset [c]_\emptyset \langle b \rangle_\emptyset \mathtt{tt}$$

to be checked on the trace $[cab]$. *It holds that:*

$\mathcal{D}_{a,\emptyset}(cab) = [\bar{c}b]$, *for the* Σ_1, *while* $\mathcal{D}_{a,\emptyset}(cab) = [\bar{c}\bar{b}]$, *for* Σ_2.

Then, after the cleaning,

$\mathcal{D}_{c,\emptyset}(cb) = [\bar{b}]$, *and*

$\mathcal{D}_{c,\emptyset}(cb) = [b]$.

Finally, again after the cleaning,

$\mathcal{D}_{b,\emptyset}(\epsilon) = null$, *and*

$\mathcal{D}_{b,\emptyset}(b) = \epsilon$,

thus $[cab] \models \varphi_3$ *for* Σ_2, *but* $[cab] \not\models \varphi_3$ *for* Σ_1; *in fact, the marked event* $\bar{b}$ *does not occur after* c *in any run.*

The satisfaction of an LTSC formula by a trace system is defined as follows:

Definition 6. *Let $TS = ((A, D), T)$ be a trace system.*

$$
\begin{aligned}
&TS \models \mathtt{tt} \\
&TS \not\models \mathtt{ff} \\
&TS \models \psi_1 \vee \psi_2 \quad \textit{iff} \quad TS \models \psi_1 \textit{ or } TS \models \psi_2 \\
&TS \models \psi_1 \wedge \psi_2 \quad \textit{iff} \quad TS \models \psi_1 \textit{ and } TS \models \psi_2 \\
&TS \models [a]_S\, \psi \quad \textit{iff} \quad \forall \tau \in T,\ \tau \models [a]_S\, \psi \\
&TS \models \langle a \rangle_S\, \psi \quad \textit{iff} \quad \exists \tau \in T,\ \tau \models \langle a \rangle_S\, \psi \\
&TS \models \nu Z.\psi \quad \textit{iff} \quad TS \models \nu Z^n.\psi \textit{ for all natural numbers } n \\
&TS \models \mu Z.\psi \quad \textit{iff} \quad TS \models \mu Z^n.\psi \textit{ for some natural number } n
\end{aligned}
$$

Example 7. *Now reconsider the expression of Example 3:*

$$e = rec(a.rec(d))$$

and suppose having to check on e the formula:

$$\varphi_4 = \nu Z.[d]_\emptyset \langle a \rangle_\emptyset Z$$

Since $[\epsilon]$ satisfies any formula, $e \models \nu Z^n.[d]_\emptyset \langle a \rangle_\emptyset Z, \forall n$, if $[a \lfloor\ \rfloor_{T_0}], [ad \lfloor\ \rfloor_{T_3} \lfloor\ \rfloor_{T_0}]$ satisfy $\nu Z^n.[d]_\emptyset \langle a \rangle_\emptyset Z$ ($a \lfloor\ \rfloor_{T_0}$ and $ad \lfloor\ \rfloor_{T_3} \lfloor\ \rfloor_{T_0}$ are the one-unfoldings of $\lfloor\ \rfloor_{T_0}$).

Since $\mathcal{D}_{d,\emptyset}(\mathcal{C}l^2(a \lfloor\ \rfloor_{T_0})) = null$, we have to verify that:

$$\mathcal{D}_{d,\emptyset}(\mathcal{C}l^2(ad \lfloor\ \rfloor_{T_3} \lfloor\ \rfloor_{T_0})) = [\lfloor\ \rfloor_{T_3} \lfloor\ \rfloor_{T_0}] \models \langle a \rangle_\emptyset [\nu Z^{n-1}.[d]_\emptyset \langle a \rangle_\emptyset Z]$$

Since $\mathcal{D}_{a,\emptyset}(\mathcal{C}l^1(a, d \lfloor\ \rfloor_{T_3} a \lfloor\ \rfloor_{T_0})) = [\lfloor\ \rfloor_{T_0}]$, and:

$$[\lfloor\ \rfloor_{T_0}] \models \nu Z^{n-1}.[d]_\emptyset \langle a \rangle_\emptyset Z$$

then:

$$[\lfloor\ \rfloor_{T_0}] \models \nu Z^n.[d]_\emptyset \langle a \rangle_\emptyset Z, \forall n$$

4. Transformation Rules to Obtain Abstract Trace Systems

In this section, we present a syntactic transformation algorithm, which, given a set ρ of events and an expression e, transforms e into e', where both e and e' satisfy the same set of LTSC formulae with events occurring in ρ. In general, the trace system corresponding to e' is smaller than the one corresponding to e. Our aim is two-fold: given a formula, to find a suitable set ρ and, given ρ, to eliminate from an expression a suitable superset of ρ. A suitable ρ depending on a formula is the following.

Occurring Events: $\mathcal{O}(\varphi)$**.**Given an LTSC formula φ, $\mathcal{O}(\varphi)$ is the union of all the events α and the sets S appearing in the modal operators ($[\alpha]_S\psi$, $\langle\alpha\rangle_S\psi$) occurring in φ.

Definition 7 (Transformation rule)**.** *Let $\Sigma = (A, D)$ be a concurrent alphabet and $\rho \subseteq A^{-}$ and e an expression over A^{-}. We define $\mathcal{T}_\rho(e)$ as:*

$$\mathcal{T}_\rho(\alpha) = \begin{cases} nil & \alpha \notin \rho \\ \alpha & \alpha \in \rho \end{cases}$$

$$\begin{aligned} \mathcal{T}_\rho(e_1.e_2) &= \mathcal{T}_\rho(e_1).\mathcal{T}_\rho(e_2) \\ \mathcal{T}_\rho(e_1 + e_2) &= \mathcal{T}_\rho(e_1) + \mathcal{T}_\rho(e_2) \\ \mathcal{T}_\rho(e_1 \| e_2) &= \mathcal{T}_{\rho'}(e_1) \| \mathcal{T}_{\rho'}(e_2) \text{ where } \rho' = \rho \cup (alph(e_1) \cap alph(e_2)) \\ \mathcal{T}_\rho(rec(e)) &= rec(\mathcal{T}_\rho(e)) \end{aligned}$$

The previous rule maintains in the traces events belonging to a suitable superset of $\mathcal{O}(\varphi)$: in fact, besides the occurring events of the formula, it maintains also the communication events .

The complexity of the transformation operator $\mathcal{T}$ is linear in the length of the specification. This result, together with Definition 6 of the satisfaction on traces, further reduces the complexity of the model checking of temporal logic formulae.

The following theorem deals with the abstraction of a trace system induced by the formula φ and obtained in two steps: the first step syntactically reduces the event expression; the second one reduces the trace system, which is the semantics of the expression.

Now, we extend the definition of projection to trace systems.

Definition 8. *Let $TS = ((A, D), T)$ be a trace system and $B \subseteq A$;*

$$\Pi_B(TS) = ((B \cup \{\lfloor\ \rfloor_x \mid \lfloor\ \rfloor_x \in A\}, \Pi_B(D)), \Pi_B(T))$$

where:

(1) $\Pi_B(D) = \{(a, b) \mid (a, b) \in D$ *and* $a, b \in B\}$*;*

(2) $\Pi_B(T) = \{\Pi_B(w) \mid w \in T\}$*, and* $\forall \lfloor\ \rfloor_x \in A, x = \{\Pi_B(w) \mid w \in x\}$.

Theorem 1. *Consider an expression e and an LTSC formula ψ.*

$$TS(e) \models \psi \quad \textit{if and only if} \quad \Pi_{\mathcal{O}(\psi)}(TS(\mathcal{T}_{\mathcal{O}(\psi)}(e))) \models \psi$$

Proof: see the Appendix.

Example 8. *Consider the following expression:*

$$e = rec(a.(b \| c') \parallel c'.a'.(b' \| c))$$

and try to prove the properties:
$\varphi_1 = \nu Z.\langle a \rangle_\emptyset\, Z$*: "there exists a run in which the event a, preceded by any event, can always occur".*
$\varphi_2 = [a']_{\{a\}}$ `ff`*: "it is not possible to perform a' if a has not occurred before".*
By our methodology, we have to check:

- $TS(e) \models \varphi_1$, *and*
- $TS(e) \models \varphi_2$

through the checking of (see Theorem 1):

- $\Pi_{\rho_1}(TS_1(\mathcal{T}_{\rho_1}(e))) \models \varphi_1$, *with* $\rho_1 = \mathcal{O}(\varphi_1) = \{a\}$
- $\Pi_{\rho_2}(TS_2(\mathcal{T}_{\rho_2}(e))) \models \varphi_2$, *with* $\rho_2 = \mathcal{O}(\varphi_2) = \{a, a'\}$

The transformation rules applied over e *with* ρ_1 *obtain:*

$\mathcal{T}_{\rho_1}(e)$

= { *applying Definition 7 and the rules set for the operators in Section 2.1*}

$rec(a.c' \parallel c')$

While, using ρ_2*, similarly we obtain:* $\mathcal{T}_{\rho_2}(e) = rec(a.c' \parallel c'.a')$.

The trace language of $TS_1(rec(a.c' \parallel c'))$ *is*

$$\{[\epsilon], \lfloor\ \rfloor_{T_3}\},\ \textit{where}\ T_3 = \{[ac']\}\ \textit{is the trace language of}\ TS_3(a.c' \parallel c').$$

The trace language of $TS_2(rec(a.c' \parallel c'.a'))$ *is:*

$$\{[\epsilon], \lfloor\ \rfloor_{T_4}\},\ \textit{with}\ T_4 = \{[ac'a']\}\ \textit{that is the trace language of}\ TS_4(a.c' \parallel c'.a').$$

Finally, by applying Definition 6, we can prove that $\Pi_{\rho_1}(TS_1) \models \varphi_1$ *and* $\Pi_{\rho_2}(TS_2) \models \varphi_2$*. In the first case, the trace* $[\lfloor\ \rfloor_{T_3}]$ *can be unfolded (the first one-unfolding is* $[ac' \lfloor\ \rfloor_{T_3}]$*) step by step and simplified for the successive step (the next one-unfolding for the simplified trace produces the trace* $[c'ac' \lfloor\ \rfloor_{T_3}]$*). We can see that, at each step n, the formula is verified on:*

$$\nu Z^0.\varphi = \texttt{tt}$$
$$\nu Z^{n+1}.\varphi = \varphi[\nu Z^n.\varphi/Z]$$

Consequently, $\varphi_1 = \nu Z.\langle a\rangle_\emptyset\ Z$ *holds on* e*. For* $\varphi_2 = [a']_{\{a\}}\,\texttt{ff}$*, both traces should verify the formula: it is easy to see that, for both* $[\epsilon]$ *and* $[ac'a' \lfloor\ \rfloor_{T_4}]$ *(the one-unfolding of* $\lfloor\ \rfloor_{T_4}$*),* φ_2 *holds.*

5. Conclusions and Related Works

In this work, we define the notion of satisfaction for the formulae of a temporal logic calculus, when interpreted on a trace system, *i.e.*, a partial order representation of the concurrent system. The calculus we use is a temporal logic, whose formulae directly characterize possible abstractions of the system preserving their truth values; more precisely, the events relevant for checking a formula are only the ones explicitly mentioned. Different action logics, such as [14], whose operators could be used in a linear fashion to concisely express fairness properties, are not suitable to individuate system abstractions preserving the truth values of formulae. We use a well-known partial order model for computations, that is Mazurkiewicz's trace system [20]. Recently, Mazurkiewicz traces have been also extended with time in order to capture the concurrency and timing constraints among the services of systems [23].

It is known that model checking has a different complexity when using linear time or branching time logics [5,17,18]. For example, given a transition system of size n and an alternation-free temporal logic formula of size m, model checking algorithms for the branching time logic (CTL) run in time $\mathcal{O}(nm)$, while those ones for the linear time logics (LTL) run in time $\mathcal{O}(n2^m)$.

The works [24,25] relate branching and linear model checking. The authors study the problem of deciding whether a linear-time property, specified by either an automaton or an LTL formula, can be translated to a branching time logic formula and describe the translation when this exists. A disadvantage of this method is that a higher complexity of the formula is achieved, since its size must be increased by some prefix; moreover, also fairness requirements must be expressed by a new formula prefixed to the translation of the old one. Other works [26,27] check linear time formulae on a structure that is the unfolding of the Petri net representing the system, *i.e.*, a partial order model. Nevertheless, this method requires the construction of a, possibly infinite, safe net to represent the behavior of the system. Solutions exist to this problem that exploit McMillan's finite prefix [28]. The product automaton is obtained from the finite prefix and the Buchi automaton representing the formula. In [29,30], for example, the satisfaction of linear time mu-calculus formulae is checked using alternating Buchi automata. Some different approaches (for example, [20,31,32]) interpret the logic on the dependence graphs among events, instead of linearizations. Moreover, the state explosion problem is present also for this type of graph. A similar approach is the automata-theoretic one [25], which is based on the product between the finite state automaton representing the system and the one representing the formula. Furthermore, such an approach suffers from the state-explosion problem, due to the interleaving of concurrent events; several methods for the reduction of the state space have been suggested in this case [7,9,33,34].

A different approach could be that of using a partial order interpretation, where it is possible to distinguish concurrency from nondeterminism, trying to keep the advantage of not exploding the state space of the system due to the concurrency using trace systems. Our method exploits the compact notation of equivalence classes of behaviors; moreover, the satisfaction of a formula is decided by an algorithm that works directly on the traces (thus, a sequential memory representation) without constructing any type of graph; so saving space and time. The work pays further attention to the reduction of the state space of the system, since, as argued in [17,24], this is the greatest part of the complexity of the model checking procedure, provided that interesting properties are generally of a small size. For this purpose, we use selective mu-calculus to describe system properties, and trace systems are reduced by eliminating the events that do not alter the truth value of a given formula. Finally, it is worth noting that a lot of interesting properties, such as mutual exclusion and absence of starvation, can be elegantly expressed by linear time formulae that usually use a partial order interpretation; while the interleaving interpretation does not allow an easy expression of fairness properties; nevertheless, the use of the selective mu-calculus allows us to express properties, such as precedence and fairness, in a very compact way; for example, the fairness property "in all computation, eventually aoccurs" can be expressed through selective mu-calculus by the formula $\mu Z.(\langle - \rangle_{\emptyset}\, \mathtt{tt} \wedge [-a]_{\emptyset}\, Z)$.

The works [9,10,34,35] also follow the partial order approach to model checking and consider only a representative among all interleavings of actions generated by a parallel composition. The properties that are well handled by these approaches do not concern precedence relations between actions, while they can be profitably used to prove, for example, deadlock freedom. In our approach, properties concerning

precedence relations between actions are, in general, described by formulae that induce a consistent reduction of the trace system. On the contrary, the formula describing deadlock freedom induces no reduction, since it involves all events. Thus, the two approaches can be considered as complementary. Other approaches use symbolic model checking (for a survey on the use of this formal verification technique for linear temporal logic, see [36]), which is particularly suited to the verification of reactive systems or concurrent programs. In these approaches, the focus of the state explosion problem is shifted from the size of the state space to the size of the BDD representation, maintaining all problems of defining an automaton, recognizing the complement of the formula to be verified.

Abstraction is another successful technique for fighting the state explosion problem in model checking. In [37], the authors present a novel game-based approach to abstraction-refinement for the full μ-calculus, interpreted over three-valued semantics, successively improved in [38].

Pushdown systems are transition systems whose states include a stack of unbounded length; hence, they are strictly more expressive than finite state systems. One can argue that pushdown systems are a natural model for sequential programs with procedures where there is no restriction on the call hierarchy among the procedures. Arbitrary recursion is allowable, since the stack can keep track of active procedure calls. Differently form our approach, the main restriction of this kind of model is that it does not handle parallelism. In fact, in [39], the authors show applications of their model checking algorithm for pushdown systems only in the area of sequential program analysis. The model checking problem of pushdown systems against standard branching temporal logics has been intensively studied in the literature. It has been confirmed that, with pushdown systems, the model checking problem is much harder for branching-time temporal logics than for linear-time temporal logics. In particular, for the modal mu-calculus, the most powerful branching temporal logic used for verification, the problem is known to be EXPTIME-complete (even for a fixed formula) [40]. The problem remains EXPTIME-complete also for the logic, CTL. In [41], the author shows that the complexity of the pushdown model checking problem for CTL* is, in fact, 2EXPTIME-complete. A natural generalization of pushdown machines is pushdown machines with more than one stack [42]. This generalization, unfortunately, is not smooth in terms of the power of these machines: a pushdown automaton with two or more stacks is known to recognize all recursively enumerable languages. The model in its full generality is, thus, intractable. However, for certain model checking applications, pushdown automata with two or more stacks are useful. In [43], the model checking problem for specifications given by nondeterministic pushdown tree automata is studied. The author consider both finite-state (regular) and infinite-state (non-regular) systems. It is shown that for finite-state systems, the model checking problem is solvable in time exponential in both the system and the specification. On the other hand, the model checking problem for context-free systems is undecidable; already for a weak type of pushdown tree automata.

In [44], the authors consider the sabotage modal logic (SML) [45], which can arbitrarily delete edges of the model; thus, it has the ability to modify the model under evaluation. In [44], it was shown that the sabotage modality already strengthens modal logic in such a way that all nice model-theoretic properties and algorithmic complexities get lost. In fact, from the viewpoint of complexities, SML much more resembles first-order logic than modal logic (with the exception that the formula complexity remains in PTIME).

In [46], the authors examine the complexity of the module checking problem (*i.e.*, model checking of open systems) for linear and branching temporal logics. They prove that the problem of module checking is EXPTIME-complete for specifications in CTL and 2EXPTIME-complete for specifications in CTL*. In [47], module checking has been extended to a setting where the environment has imperfect information about the state of the system, *i.e.*, to the case where the environment has only a partial view of the system's control states (see also [48] for related work regarding imperfect information). Recently, [49,50] extended model checking of pushdown systems by introducing open pushdown systems (with perfect information) that interact with their environment. It is shown in [49,50] that CTL pushdown module checking is 2EXPTIME-complete and, thus, much harder than pushdown model checking. In [51], the authors extend pushdown module checking to the imperfect information setting and pushdown store content. They study the complexity of this problem with respect to the branching-time temporal logics, CTL, CTL* and the propositional μ-calculus. They show that pushdown module checking becomes undecidable when the environment has imperfect information.

Improved model checking algorithms have been developed also for hierarchical systems [52–54]. Such systems are exponentially more succinct than standard state transition graphs, as repeated sub-systems are described only once.

In conclusion, our method consists of a variety of attacks to the complexity of the verification.

(1) We work directly and only on traces to perform model checking, without representing either traces or systems by some sort of graph, so saving memory.
(2) We obtain a finite trace system, also when using unbounded iteration. In such a way, we can perform model checking also in the presence of infinite computations.
(3) We reduce the dimension of the trace system, in the number of traces and in the number of events in each trace. Beside the use of abstraction to reduce the number of events of the initial traces, we maintain in a trace at each verification step only the events useful for the following steps, so performing a kind of on-the-fly verification. In such way, we save space, but also verification time, since we decrease the number of times a formula has to be checked.
(4) We manipulate traces to decide the satisfaction on a single trace with a polynomial complexity depending on the dimension of the formula and of the trace. The precise complexity of the method needs a deeper examination, since, in general, it depends on the level of concurrency of the systems and on the number of the rec operators and of the possible one-unfoldings of each hole.

Author Contributions

Antonella Santone and Gigliola Vaglini are both responsible for the concept of the paper, the results presented and the writing. Both authors have read and approved the final published manuscript.

Appendix

Proof of Theorem 1

We first give some technical lemmas.

Lemma 1. *Let* $TS_1 = ((A_1, D_1), T_1)$ *and* $TS_2 = ((A_2, D_2), T_2)$ *be two trace systems,* $A_1 \cap A_2 = \emptyset$ *and* $\rho \subseteq A_1 \cup A_2$.

(1) $\Pi_\rho(TS_1.TS_2) = \Pi_\rho(TS_1).\Pi_\rho(TS_2)$

(2) $\Pi_\rho(TS_1 + TS_2) = \Pi_\rho(TS_1) + \Pi_\rho(TS_2)$

Proof.

Item 1. $\Pi_\rho(TS_1.TS_2)$

$=$ { *by definition of the concatenation of trace systems* }
$\Pi_\rho((A_1 \cup A_2, D_1 D_2^\circ), \{\tau_1.\tau_2 \mid \tau_1 \in T_1,\ \tau_2 \in T_2\})$

$=$ { *by Definition 8* }
$((\rho, \Pi_\rho(D_1 D_2^\circ)), \Pi_\rho(\{\tau_1.\tau_2) \mid \tau_1 \in T_1,\ \tau_2 \in T_2\})$

$=$ { *by the properties of projection over strings and over dependencies* }
$((\rho, \Pi_\rho(D_1)\Pi_\rho(D_2)^\circ), \{\Pi_\rho(\tau_1) \mid \tau_1 \in T_1\}.\{\Pi_\rho(\tau_2) \mid \tau_2 \in T_2\}$

$=$ { *by Definition 8* }
$\Pi_\rho(TS_1).\Pi_\rho(TS_2)$

Item 2. *This case can be proven in a similar way.*

In the following, given a trace language T, by $alph(T)$, we denote the set of symbols occurring in all traces belonging to T.

Lemma 2. *Let* $TS_1 = ((A_1, D_1), T_1)$ *and* $TS_2((A_2, D_2), T_2)$ *be two trace systems,* $\rho, \rho' \subseteq A_1 \cup A_2$ *and* $\rho \subseteq \rho'$.

(1) $\Pi_\rho(TS_1 \| TS_2) = \Pi_\rho(\Pi_{\rho'}(TS_1) \| \Pi_{\rho'}(TS_2))$
where $\rho' = \rho \cup (alph(T_1) \cap alph(T_2))$

(2) $\Pi_\rho(TS_1^i) = (\Pi_\rho(TS_1))^i$

Proof.

Item 1. *First, we prove, ad absurdum, that:* $\Pi_\rho(T_1 \| T_2) \subseteq \Pi_\rho(\Pi_{\rho'}(T_1) \| \Pi_{\rho'}(T_2))$

Suppose that $\tau \notin \Pi_\rho(\Pi_{\rho'}(T_1) \| \Pi_{\rho'}(T_2))$.
$\tau \notin \Pi_\rho(\Pi_{\rho'}(T_1) \| \Pi_{\rho'}(T_2))$

implies { *by Definition 8.2* }
$\tau \notin \{\Pi_\rho(w) \mid w \in (\Pi_{\rho'}(T_1) \| \Pi_{\rho'}(T_2)\}$

implies { *by definition of the parallel composition of trace languages* }
$\tau \notin \{\Pi_\rho(w) \mid w \in \{w' \mid\ \Pi_{A_1}(w') \in \Pi_{\rho'}(T_1)$ *and* $\Pi_{A_2}(w') \in \Pi_{\rho'}(T_2)\}\}$

implies:
$\tau = \Pi_\rho(w)$ *and* $\Pi_{A_1}(w) \in \Pi_{\rho'}(T_1)$ *or* $\Pi_{A_2}(w) \in \Pi_{\rho'}(T_2)$

implies: { *by Definition 8.2* }
$\tau = \Pi_\rho(w)$ *and* $(\Pi_{A_1}(w) \notin \{\Pi_{\rho'}(k) \mid k \in T_1\}$ *or* $\Pi_{A_2}(w) \notin \{\Pi_{\rho'}(k) \mid k \in T_1\})$

implies:

$\tau = \Pi_\rho(w)$ *and* $((\Pi_{A_1}(w) = \Pi_{\rho'}(k)$ *and* $k \notin T_1)$ *or* $(\Pi_{A_2}(w) = \Pi_{\rho'}(k)$ *and* $k \notin T_2))$

absurdum { *since* $\rho' = \rho \cup (alph(T_1) \cap alph(T_2))$ }

The case $\Pi_\rho(\Pi_{\rho'}(T_1) \| \Pi_{\rho'}(T_2)) \subseteq \Pi_\rho(T_1 \| T_2)$ *can be proven similarly. The thesis holds by properties of projection over dependencies.*

Item 2. *This is similar.*

Lemma 3. *Let* s *be an expression over* A *and* $\rho \subseteq A$.

$$\Pi_\rho(TS(s)) = \Pi_\rho(TS(\mathcal{T}_\rho(s)))$$

Proof.

The proof is made by the induction on the structure of the term.

Base step. $s = nil$*: straightforward.*

Inductive step. *We denote* $TS(a) = (\{a\}, \{(a, a)\}), \{[a]\})$.

$\underline{s = a.s_1}$*:*

$\Pi_\rho(TS(a.s_1))$

$=$ { *by Definition 2 and Lemma 1.(1)* }

$\Pi_\rho(TS(a)).\Pi_\rho(TS(s_2))$

$=$ { *by the inductive hypothesis and Lemma 1.(1)* }

$\Pi_\rho(TS(\mathcal{T}_\rho(a)).TS(\mathcal{T}_\rho(s_1)))$

$=$ { *by Definition 2* }

$\Pi_\rho(TS(\mathcal{T}_\rho(a.s_1)))$

All other cases can be proven in a similar way using Item 2 *of Lemmas 1 and 2.*

Lemma 4. *Let* ψ *be a selective mu-calculus formula, with* $\mathcal{O}(\psi) = \rho$ *and* e *an expression.*

$$TS(e) \models \psi \textit{ if and only if } \Pi_\rho(TS(e)) \models \psi$$

Proof.

The proof is made by induction on the structure of the formula.

Base step. $\psi = \mathtt{tt}, \mathtt{ff}$*: straightforward.*

Inductive step.

$\underline{\psi = \langle\alpha\rangle_S \ \psi'}$*: Suppose that:*

$TS(e) = ((A, D), T)$ *then* $\Pi_\rho(TS(e)) = ((\rho, \Pi_\rho(D)), \Pi_\rho(T))$.

$TS(e) \models \langle\alpha\rangle_S \psi'$

iff { *by Definition 6* }

$\exists [s] \in T.[s] \models \langle\alpha\rangle_S \psi'$

iff { *by Definition 5* }

$\exists [s] \in T,\ \exists \sigma' \in \widehat{s}$, *such that* $\mathcal{D}_{\alpha,S}(\mathcal{C}l^1(\alpha,\sigma')) \neq null$ *and* $\mathcal{D}_{\alpha,S}(\mathcal{C}l^1((\alpha,\sigma'))) \models \psi'$

iff, {*since* $\mathcal{D}_{\alpha,S}(\mathcal{C}l^1(\alpha,\sigma')) = \mathcal{D}_{\alpha,S}(\mathcal{C}l^1(\alpha,\Pi_\rho(\sigma')))$ *and* $\Pi_\rho([\sigma']) \in \Pi_\rho(T)$ }

$\exists \Pi_\rho([s]) \in \Pi_\rho(T), \exists \Pi_\rho(\sigma') \in \Pi_\rho(\widehat{s})$, *such that* $\mathcal{D}_{\alpha,S}(\mathcal{C}l^1(\alpha,\Pi_\rho([\sigma']))) \neq null$ *and* $\mathcal{D}_{\alpha,S}(\mathcal{C}l^1(\alpha,\Pi_\rho([\sigma']))) \models \psi'$

iff { *by Definition 5* }

$\exists\, \Pi_\rho([\sigma']) \in \Pi_\rho(T).\Pi_\rho([\sigma']) \models \langle\alpha\rangle_S\, \psi'$

iff { *by Definition 6* }

$\Pi_\rho(TS(e)) \models \langle\alpha\rangle_S\, \psi'$

$\psi = [\alpha]_S\ \psi'$: *this case can be proven in a similar way.*

The proofs of all other cases follow by a symmetric argument and by inductive hypothesis.

Now, we are ready to prove the main theorem.

THEOREM 1. *Let* e *be an expression and* ψ *a selective mu-calculus formula.*

$$TS(e) \models \psi \quad \textit{if and only if} \quad \Pi_{\mathcal{O}(\psi)}(TS(\mathcal{T}_{\mathcal{O}(\psi)}(e))) \models \psi$$

Proof.
Let $\rho = \mathcal{O}(\psi)$.

$TS(e) \models \psi$

iff { *by Lemma 4* }
$\Pi_\rho(TS(e)) \models \psi$

iff { *by Lemma 3* } $\Pi_\rho(TS(\mathcal{T}_\rho(e))) \models \psi$

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Manna, Z.; Pnueli, A. The anchored version of the temporal framework. In Proceedings of the Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, 30 May–3 June 1988; Lecture Notes in Computer Science, Volume 354, pp. 201–284.
2. Emerson, E.A.; Srinivasan, J. Branching time temporal logic. In Proceedings of the Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, 30 May–3 June 1988; Lecture Notes in Computer Science, Volume 354, pp. 123–172.

3. Mazurkiewicz, A. Basic notions of Trace Theory. In Proceedings of the Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, 30 May–3 June 1988; Lecture Notes in Computer Science, Volume 354, pp. 285–363.
4. Mazurkiewicz, A.; Ochmanski, E.; Penczek, W. Concurrent systems and inevitability. *Theor. Comput. Sci.* **1989**, *64*, 281–304.
5. Clarke, E.M.; Emerson, E.A.; Sistla, A.P. Automatic verification of finite-state concurrent systems using temporal logic verification. *ACM Trans. Program. Lang. Syst.* **1986**, *8*, 244–263.
6. Bryant, R.E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.* **1986**, *C-35*, 677–691.
7. Burch, J.; Clarke, E.; McMillan, K.; Dill, D.; Hwang, L. Symbolic Model Checking: 10^{20} States and Beyond. In Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia, PA, USA, 4–7 June 1990; pp. 428–439.
8. Clarke, E.M.; Grumberg, O.; Long, D.E. Model checking and abstraction. *Trans. Program. Lang. Syst.* **1992**, *16*, 343–354.
9. Garavel, H.; Lang, F.; Mateescu, R.; Serwe, W. CADP 2011: A toolbox for the construction and analysis of distributed processes. *Int. J. Softw. Tools Technol. Transf.* **2013**, *15*, 89–107.
10. Godefroid, P. *Partial-Order Methods for the Verification of Concurrent Systems*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1996; Volume 1032.
11. Barbuti, R.; de Francesco, N.; Santone, A.; Vaglini, G. Selective mu-calculus: New modal operators for proving properties on reduced transition systems. In Proceedings of the FORTE X/PSTV XVII '97, Osaka, Japan, 18-21 November 1997; Chapman & Hall: London, UK, 1997; pp. 519–534.
12. Barbuti, R.; de Francesco, N.; Santone, A.; Vaglini, G. Selective mu-calculus and formula-based equivalence of transition systems. *J. Comput. Syst. Sci.* **1999**, *59*, 537–556.
13. Stirling, C. An Introduction to Modal and Temporal Logics for CCS. In Proceedings of the UK/Japan Workshop on Concurrency : Theory, Language, and Architecture, Oxford, UK, 25–27 September 1989; Lecture Notes in Computer Science, Volume 391.
14. De Nicola, R.; Vaandrager, F.W. Action *versus* State based Logics for Transition Systems. In Proceedings of the LITP Spring School on Theoretical Computer Science on Semantics of Systems of Concurrent Processes, La Roche Posay, France, 23–27 April 1990; Lecture Notes in Computer Science, Volume 469, pp. 407–419.
15. Mazurkiewicz, A. Trace Theory. Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8–19. September 1986. Lecture Notes in Computer Science, Volume 255, 1987; pp. 279–324.
16. Bradfield, J.; Stirling, C. Local model checking for infinite state spaces. *Theor. Comput. Sci.* **1992**, *96*, 157–174.
17. Lichtenstein, O.; Pnueli, A. Checking that finite state concurrent programs satisfy their linear specification. In Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '85), New Orleans, LA, USA, 14-16 January 1985; pp. 97–107.

18. Sistla, A.P.; Clarke, E.M. The complexity of propositional linear time logics. *J. ACM* **1985**, *32*, 733–749.
19. Godefroid, P.; Piterman, N. LTL Generalized Model Checking Revisited. In Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '09), Savannah, GA, USA, 18–20 January 2009; Lecture Notes in Computer Science, Volume 5403, pp. 89–104.
20. Gastin, P.; Petit, A. *The Book of Traces, Chapter Infinite Traces*; Diekert, V., Rozenberg, G., Eds.; World Scientific: Singapore, Singapore, 1995.
21. Penczek, W. Temporal logics for trace systems: On automated verification. *Int. J. Comput. Sci.* **1993**, *4*, 31–67.
22. Bradfield, J. The modal mu-calculus alternation hierarchy is strict. In Proceedings of the 7th International Conference CONCUR'96, Pisa, Italy, 26–29 August 1996; Volume 1119; pp. 233–246.
23. Chieu, D.V.; Hung, D.V. An extension of Mazukiewicz traces and their applications in specification of real-time systems. In Proceedings of the Second International Conference on Knowledge and Systems Engineering (KSE '10), Hanoi, The Netherlands, 7–9 October 2010; pp. 167–171.
24. Kupferman, O.; Vardi, M.Y. Relating Linear and Branching Model Checking. In Proceedings of the IFIP TC2/WG2.2, 2.3 International Conference on Programming Concepts and Methods (PROCOMET '98), Shelter Island, NY, USA, 8–12 June1998; IFIP-Chapman-Hall: London, UK.
25. Kupferman, O.; Vardi, M.Y. Freedom, weakness, and determinism: From linear-time to branching-time. In Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS '98), Indianapolis, IN, USA, 21–24 June 1998; IEEE Computer Society: Washington, DC, USA, 1998.
26. McMillan, K.L. Trace theoretic verification of asynchronous circuits using unfoldings. In Proceedings of the 7th International Conference on Computer-Aided Verification (CAV '95), Liege, Belgium, 3–5 July 1995; Lecture Notes in Computer Science, Volume 939, pp. 180–195.
27. Wallner, F. Model checking LTL using net unfoldings. In Proceedings of the 10th International Conference on Computer-Aided Verification (CAV '98), Vancouver, BC, Canada, 28 June–2 July 1998; Lecture Notes in Computer Science, Volume 1427, pp. 207–218.
28. McMillan, K.L. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In Proceedings of the 4th International Workshop on Computer-Aided Verification (CAV '92), Montreal, QC, Canada, 29 June–1 July 1992; Lecture Notes in Computer Science, Volume 663, pp. 164–174.
29. Bollig, B.; Leucker, M. Deciding LTL over Mazurkiewicz Traces. In Proceedings of the Symposium on Temporal Representation and Reasoning (TIME '01), Cividale, Italy, 14–16 June 2001; IEEE Computer Society Press: Washington, DC, USA, 2001.
30. Kaivola, R. A simple decision method for the linear time mu-calculus. In Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), Berlin, Germany, 11–13 May 1995; Workshops in Computing; Springer: London, UK, 1995; pp. 190–204.

31. Thiagarajan, P.S.; Walukiewicz, I. An Expressively Complete Linear Time Temporal Logic for Mazurkiewicz Traces. In Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97), Warsaw, Poland, 29 June–2 July 1997; IEEE Computer Society: Washington, DC, USA, 1997; pp. 183–194.
32. Walukiewicz, I. *Local Logics of Traces*; BRICS Report RS-00-2; BRICS: Aarhus, Denmark, 2000.
33. Kesten, Y.; Pnueli, A.; Raviv, L. Algorithmic Verification of Linear Temporal Logic Specifications. In Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP '98), Aalborg, Denmark, 13–17 July 1998; Lecture Notes in Computer Science, Volume 1443, pp. 1–16.
34. Peled, D. All from one, one from all: On model checking using representatives. In Proceedings of the 5th International Conference on Computer-Aided Verification, (CAV '93), Elounda, Greece, 28 June–1 July 1993; Lecture Notes in Computer Science, Volume 697, pp. 409–423.
35. Dumas, X.; Boniol, F.; Dhaussy, P.; Bonnafous, E. Context Modelling and Partial-Order Reduction: Application to SDL Industrial Embedded Systems. In Proceedings of the IEEE Fifth International Symposium on Industrial Embedded Systems (SIES '10), Trento, Italy, 7–9 July 2010; pp. 197–200.
36. Rozier, K.Y. Linear temporal logic symbolic model checking. *Comput. Sci. Rev.* **2011**, *5*, 163–203.
37. Grumberg, O.; Lange, M.; Leucker, M.; Shoham, S. When not losing is better than winning: Abstraction and refinement for the full mu-calculus. *Inf. Comput.* **2007**, *205*, 1130–1148.
38. Fecher, H.; Shoham, S. Local abstraction-refinement for the μ-calculus. *Softw. Tools Technol. Transf.* **2011**, *13*, 289–306.
39. Esparza, J.; Hansel, D.; Rossmanith, P.; Schwoon, S. Efficient Algorithms for Model Checking Pushdown Systems. In Proceedings of the 12th International Conference on Computer-Aided Verification (CAV '00), Chicago, IL, USA, 15–19 July 2000; Lecture Notes in Computer Science, Volume 1855, pp. 232–247.
40. Walukiewicz, I. Pushdown processes: Games and Model Checking. In Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96), New Brunswick, NJ, USA, 31 July–3 August 1996; Springer-Verlag: Berlin/Heidelberg, Germany, 1996; Volume 1102, pp. 62–74.
41. Bozzelli, L. Complexity results on branching-time pushdown model checking. *Theor. Comput. Sci.* **2007**, *379*, 286–297.
42. Carotenuto, D.; Murano, A.; Peron, A. 2-Visibly Pushdown Automata. In Proceedings of the 11th International Conference on Developments in Language Theory (DLT '07), Turku, Finland, 3–6 July 2007; pp. 132–144.
43. Kupferman, O.; Piterman, N.; Vardi, M.Y. Pushdown Specifications. In Proceedings of the 9th International Conference, LPAR, Tbilisi, Georgia, 14–18 October 2002; pp. 262–277.
44. Löding, C.; Rohde, P. Model Checking and Satisfiability for Sabotage Modal Logic. In Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '03), Mumbai, India, 15–17 December 2003; pp. 302–313.

45. Benthem, J.V. An essay on sabotage and obstruction. In *Festschrift in Honour of Jörg Siekmann, LNAI*; Hutter, D., Werner, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2002.
46. Kupferman, O.; Vardi, M.Y.; Wolper, P. Module checking. *Inf. Comput.* **2001**, *164*, 322–344.
47. Kupferman, O.; Vardi, M.Y. Module checking revisited. In Proceedings of the 9th International Conference (CAV97), Haifa, Israel, 22–25 June 1997; Springer-Verlag: Berlin/Heidelberg, Germany, 1997; Lecture Notes in Computer Science, Volume 1254, pp. 36–47.
48. Chatterjee, K.; Doyen, L.; Henzinger, T.A.; Raskin, J. Algorithms for omega-regular games with imperfect information. *Log. Methods Comput. Sci.* **2007**, *3*, 1–23.
49. Bozzelli, L.; Murano, A.; Peron, A. Pushdown module checking. In Proceedings of the 12th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR '05), Montego Bay, Jamaica, 2–6 December 2005; Springer-Verlag: Berlin/Heidelberg, Germany, 2005; Lecture Notes in Computer Science, Volume 3835, pp. 504–518.
50. Bozzelli, L.; Murano, A.; Peron, A. Pushdown module checking, Form. *Methods Syst. Des.* **2010**, *36*, 65–95.
51. Aminof, B.; Legay, A.; Murano, A.; Serre, O.; Vardi, M.Y. Pushdown module checking with imperfect information. *Inf. Comput.* **2013**, *223*, 1–17.
52. Aminof, B.; Kupferman, O.; Murano, A. Improved model checking of hierarchical systems. *Inf. Comput.* **2012**, *210*, 68–86.
53. Alur, R.; Yannakakis, M. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.* **2001**, *23*, 273–303.
54. Alur, R.; Benedikt, M.; Etessami, K.; Godefroid, P.; Reps, T.W.; Yannakakis, M. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.* **2005**, *27*, 786–818.