Efficient online monitoring of LTL properties for asynchronous distributed systems

Thierry Massart and Cédric Meuter*

Université Libre de Bruxelles **

Abstract. We define an efficient online method to monitor the execution of asynchronous distributed systems. The code of such systems has been instrumented to record and output, during execution, some pertinent events. This output can be abstracted as a *trace* , i.e. a partially ordered set of events. During the execution, the online monitoring system collects the trace and checks on the fly that it satifies a requirement, given by any LTL property on finite sequences. The monitor checks that any execution sequence compatible with the partial order induced by the trace satisfies the property. This problem is NP-complete in the number of concurrent processes. Therefore, to provide an online monitor which, in practice, can cope in real-time with the workload, our method explores the possible configurations symbolically, as it handles sets of configurations. Moreover, it uses techniques similar to the partial order reduction, to avoid exploring as many execution interleavings as possible. It works very well in practice, compared to the standard non symbolic monitoring method.

Keywords: testing of asynchronous distributed systems, online monitoring, symbolic method, global properties, model checking of traces

1 Introduction

A distributed control system is composed of distributed hardware equipments which run concurrent processes, communicating asynchronously through some network. These communicating processes collaborate to control, through sensors and actuators, some *environment*, such as a satellite or an industrial equipment. Even if the use of an adapted software environment [1, 2] may ease the design and implementation of such a distributed reactive system, its correct building remains a non-trivial task.

Verification tools (e.g. [3–5]) may be helpful in this task. They have generally integrated some efficient exploration techniques such as partial order reduction [6, 7] or symbolic model checking [8–10] which allow them to consider some *real-size* systems. Unfortunately, quite often in practice, the *state-explosion*

* {tmassart,cmeuter}@ulb.ac.be

** Boulevard du Triomphe - CP-212, 1050 Bruxelles, Begium - Tel:+32 2 650.5603 - Fax:+32 2 650.5609

problem prevents the designer from the exhaustive verification of the complete distributed system or its model.

Therefore, several other approaches have been followed. *Testing* methods can detect system's errors before deployment. Unfortunately, it cannot guarantee its correctness. *Monitoring* techniques can detect problems during the system's execution. *Offline monitoring* is achieved after the execution on a recorded execution trace. *Online monitoring* is carried out during the system's execution; this allows both to detect immediately a (potential) error and to run recovery code to overcome it. Most modern programming languages include *assertions* and *trap* features which ease the implementation of these monitoring and recovery techniques.

Testing and offline monitoring methods have also been proposed for distributed systems to test or monitor if some *global predicates* or global temporal logic formulae are satisfied. For that purpose, the implementation (or the prototype) is instrumented to record relevant events and a special process, called *monitor*, receives these events from the various processes and checks that the observed execution satisfies the desired *global property*. A simple example of such global property is the requirement ϕ that in the distributed system S a valve A in the controlled equipment is closed before another valve B is opened and where each valve is controlled by a different distributed process. These monitoring methods can take into account that for distributed asynchronous systems, a run is generally not seen as a totally ordered sequence of events, but as a *trace*, i.e. a partially ordered set where unordered events may have occurred in any order. This causal *partial order* between the fired events are obtained through code instrumentation using, e.g., vector clocks [11, 12]. In the collected *trace*, two consecutive events of the same site are temporally ordered and communications (e.g. message transfers or shared variable manipulations) impose an order between some distributed events. *Monitoring* that an execution of the system, e.g. S given above, satisfies the *global property* ϕ reduces therefore to verifying that every sequential execution *compatible* with the partial order, satisfies ϕ or in other terms, model checking ϕ on the corresponding *trace*.

In a previous work [13], we have studied offline monitoring of distributed systems. We have shown that this problem is hard, even if the monitor is already built; and in practice, the number of compatible sequential executions may be exponential in the number of concurrent processes. We have shown that the standard partial order reduction methods are useless in this context.

However, we have kept the spirit of what is done with partial order reduction techniques, which try to minimize the exploration of execution interleavings as much as possible, and have proposed a method, efficient in practice to reduce the monitoring time. This method is symbolic since it handles sets of configurations.

In the present paper, we extend our efficient symbolic method to do *online* monitoring. We show that our method can save an exponential factor in the number of execution steps the monitor must handle after the reception of each recorded events. Therefore, in practice, it can happen that with the traditional

methods, the monitor cannot absorb the work to be done, and therefore cannot react in real time to an error, while our method works well.

This paper is organized as follows. In section 2, we detail related proposals. In section 3, we introduce our model for traces, and LTL over finite sequences. In section 4, we introduce the trace monitoring problem, and explain how it can be solved by translating a formula into a finite automaton. Then, in section 5, we present our symbolic method and we show in section 6, how this method can be refined into a symbolic exploration algorithm. Next, in section 7, we present our experimental results of various examples. Finally, further works are given in section 8.

2 Related Works

Testing and *offline* monitoring of distributed systems are studied in papers on *trace model-checking* and *global predicate detection*.

Trace model checking has been studied mainly theoretically through the definition of several linear temporal logic for Mazurkiewicz traces. A Mazurkiewicz trace [14], over an alphabet Σ with a independence relation I, can be defined as a Σ-labelled partial order set of events with special properties not explained here. For Mazurkiewicz traces, *local* and *global* trace logics have been defined. Local trace logics have been proposed in the work of Thiagaranjan on TrPTL [15] and Alur, Peled and Penczek on TLC [16]. Global trace logics include, among others, LTrL [17] proposed by Thiagarajan and Walukiewicz, and LTL on traces [18] defined by Diekert and Gastin. However, in our monitoring problem, the *trace* is an input which models a run that must be checked to see if the possible ordering of events is correct. For instance if it is required that an event a must occur before b, and if, in the trace, actions a and b are independent and can be executed in any order, the system is seen as incorrect. But, *trace temporal logics* are not "designed" to express constraints on the particular order independent actions are executed. For instance if actions a and b are independent, the trace $\mathcal{T} = ab$ expresses that a and b are concurrent. Hence, the LTL formula $a \rightarrow \Diamond b$ which expresses on semantics on sequences, that a is eventually followed by b is not so easily expressible in *trace-LTL*. So, since we do not have a priori the independence relation, these trace logics are not adapted to model-check our runs and we stick to simple sequence semantics.

Global predicate detection initially aims at answering reachability questions, i.e. does there exist a possible global configuration of the system, that satisfies a given global predicate ϕ. Garg and Chase showed in [19] that this problem is NP-complete for an arbitrary predicate, even when there is no inter-process communication. Efficient (polynomial) methods have been proposed for various classes of predicates, as *stable* predicates proposed by Chandy and Lamport [20], *independent* predicates by Charron-Bost *et al* [21], *conjunctive* predicates by Garg and Waldecker [22, 23], *linear* and *semi-linear* predicates by Chase and Garg [19] and *regular* predicates by Garg and Mittal [24]. Garg and Mittal make use of the efficient technique called *computation slicing*, to compute all cuts compatible

with a given execution satisfying a given regular predicate [25]. In [26], A. Sen and Garg present the temporal logic RCTL (for *regular*-CTL), which is a subset of the temporal logic CTL (and an extension, RCTL+). Every RCTL formula is a regular predicate; thus with RCTL formulae, we can use computation slicing to solve the predicate detection problem. In [27, 28] K. Sen et al. use an automaton to specify the system's monitor. The authors provide an explicit exploration of the state space and to limit this exploration a *window* is used. The choice of a linear temporal logic as LTL rather than a branching temporal logic as CTL seems natural since the aim is to verify that for all total orderings of the occurred events, the corresponding runs satisfy the property.

Online monitoring has been proposed by Chen et al in [29] and implemented in the Java-MOP environment. However, the properties checked are mainly local. The work by Lafortune et al [30, 31] on distributed failure diagnostic is very similar to online global predicate detection. However, contrary to us, the authors made the *synchronous assumption* that the communication between distributed processes is instantaneous. Benveniste et al [32] have proposed an algorithm to make failure diagnostic for asynchrounous systems. It is based on net unfolding [33, 34]. The diagnostic problem is defined as the work, from the description of the system and partial observations of its execution, to reconstruct all possible configurations where the system can be. Tripakis et al [35, 36] gave a theoretical analysis on the *decentralized observation and control (synthesis)* problem, where several monitors, each with only a partial observation capability, run in parallel. They showed negative (undecidable) results for this problem. In [37] K. Sen et al. define the logic PT-DTL which is a variant of past time linear temporal logic, suitable for efficient distributed online monitoring on execution traces. Note also that slight modifications [38] to the algorithms in [27, 28] allow to make them work online.

However, if it allows efficient check, neither PT-DTL of K. Sen et al nor RCTL of A. Sen and Garg can verify properties as LTL (or equivalent CTL formula) $\Box(a \rightarrow \Diamond(b \wedge c))$, i.e. every a is eventually followed by a state (or a transition) where b and c are true; formula that may be very useful during validation.

Our work uses a similar framework to what is used in [27, 28]. Since the problem is hard, we investigate here on the possibility to increase efficiency in the work of the centralized monitor: we define a *symbolic online method*, to build monitors, efficient in practice, and able to model-check if an instrumented distributed trace satisfies an LTL formula.

3 Framework

In this section, we detail our framework. We first introduce the notion of *trace* which models a run of a concurrent system. Then, we introduce LTL over finite sequences.

Trace Our runs are obtained by concurrent processes, each executing a finite sequence of variable assignments. Moreover, due to inter-process communications

(shared variable, message passing, ...), other causal dependencies are added. A run is modeled as a finite trace, i.e. a finite partially ordered set of events, where each event is labeled by the assignment which took place during this event.

Definition 1 (Trace). *For a set of variables Var, a (finite) trace $\mathcal{T}$ is a finite labeled partially ordered set $(E, \lambda, \preceq)$ where:*

- *E is a finite set of events, that can be partitioned into n subsets P_i ($1 \leq i \leq n$), one for each process.*
- *$\lambda : E \mapsto Var \times \mathbb{N}$ is a labeling function, mapping each event e to an assignment of the form $x := v$. For the event e, $\mathsf{var}(e)$ and $\mathsf{val}(e)$ denote respectively the variable x and value v of the corresponding assignment.*
- *$\preceq \subseteq E \times E$ is a partial order relation on E*

In the following, $\downarrow e$ denotes the set of predecessors of e ($\downarrow e = \{e' \mid e' \preceq e\}$) and $\uparrow e$ denotes the set of successors of e ($\uparrow e = \{e' \mid e \preceq e'\}$). We also define a *cut* C of a trace $\mathcal{T}$, which models an "execution point" of the corresponding distributed execution, as a consistent set of events $C \subseteq E$ such that $\downarrow C = C$. We note $\mathcal{C}_{\mathcal{T}}$ the set of all cuts of $\mathcal{T}$. The *set of enabled events of a cut* is defined by $\mathsf{enabled}(C) = \{e \in E \setminus C \mid \downarrow e \setminus \{e\} \subseteq C\}$. Note that for a cut C and any event $e \in \mathsf{enabled}(C)$, the set $C \cup \{e\}$ is also a cut.

As explained earlier, the system modeled as a partially-ordered trace $\mathcal{T} = (E, \lambda, \preceq)$. The semantics of such a trace is defined by the set of all finite sequences of events from E, compatible with the partial order $\preceq$.

Definition 2 (Semantics of a trace $\mathcal{T}$). *noted $[\![\mathcal{T}]\!]$ is defined by:*

$$[\![\mathcal{T}]\!] = \left\{ \sigma = e_1, ..., e_n \mid \forall 1 \leq i, j \leq n : \begin{array}{c} (\{e_i, e_j\} \in E) \wedge (e_i \preceq e_j \Rightarrow i \leq j) \\ \wedge (i \neq j \Rightarrow e_i \neq e_j) \end{array} \right\}$$

LTL **over finite sequences** Since events in a trace are single assignment, we naturally first define *basic propositions* as boolean expressions on variables of the trace. We restrict ourselves to expressions using arithmetic operators (+,-,*,/), comparison operators (<,>,=) and boolean connectors ($\wedge$, $\vee$, $\neg$). Moreover, since each trace's event is a simple assignment, each *basic proposition* uses only one variable of the trace. Example of such basic propositions are $(x = 3)$ or $((0 < 2 * x) \wedge (2 * x < 5))$. We denote by $\mathcal{P}$ the set of such basic propositions and by $\mathsf{var}(p)$, the variable appearing in a basic proposition p. The set of LTL formulas is then defined as follows: $\phi = \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \bigcirc\phi \mid \phi \,\mathcal{U}\phi$, where $\bigcirc$ is the *next state* operator, and $\mathcal{U}$ is the *until* operator. Operators $\vee$ and $\Rightarrow$ can be derived from $\vee$ and $\neg$ in a usual way. Moreover, one can use some abreviations: $\Diamond\phi \equiv \top \,\mathcal{U}\phi$ (eventually), $\Box\phi \equiv \neg\Diamond\neg\phi$ (always). A formula is interpreted over a finite sequences of events $\sigma = e_0, e_1, ..., e_n$. The satisfaction relation $\models$ is defined

on (σ, i), where σ is a sequence of events, and $0 \leq i \leq |\sigma|$, as follows:

$(\sigma, i) \models \top$ $\quad$ $(\sigma, i) \models \bigcirc\phi$ iff $(i = |\sigma|)$ or $(\sigma, i+1) \models \phi$
$(\sigma, i) \models \neg\phi$ iff $(\sigma, i) \not\models \phi$ $\quad$ $(\sigma, i) \models \phi \wedge \psi$ iff $(\sigma, i) \models \phi$ and $(\sigma, i) \models \psi$
$(\sigma, i) \models p$ $\quad$ iff $\mathsf{var}(p) = \mathsf{var}(e_i)$ and $p[\mathsf{var}(e_i) \leftarrow \mathsf{val}(e_i)]$ is true
$(\sigma, i) \models \phi\,\mathcal{U}\psi$ iff $\exists j, i \leq j \leq |\sigma|$ s.t. $(\sigma, j) \models \phi$ and $(\forall k, i \leq k < j : (\sigma, k) \models \psi)$

We note $\sigma \models \phi$ iff $(\sigma, 0) \models \phi$. In particular, $e \models p$ indicates that an event e satisfies a basic proposition p. The semantics of a formula ϕ, noted $[\![\phi]\!]$, is defined as the set of sequences satisfying ϕ. Formally, $[\![\phi]\!] = \{\sigma \mid \sigma \models \phi\}$.

4 Trace monitoring problem

Using the notations presented in the previous section, we can easily formalize the monitoring problem as follows.

Definition 3 (Trace monitoring problem (TMP)). *Given a trace $\mathcal{T} = (E, \lambda, \preceq)$ and a* LTL *formula ϕ, the trace monitoring problem (TMP) is to check whether $[\![\mathcal{T}]\!] \subseteq [\![\phi]\!]$.*

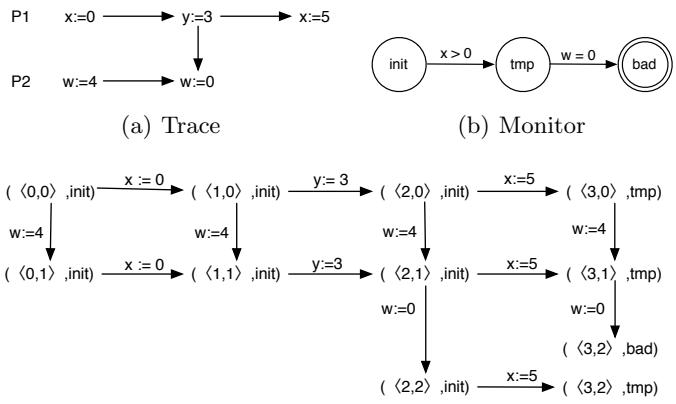
Monitors The TMP can be reduced to checking that $[\![\neg\phi]\!] \cap [\![\mathcal{T}]\!] = \emptyset$. A standard way to do that is to build a finite automaton accepting $[\![\neg\phi]\!]$. Indeed, using such an automaton, called a monitor in the following, one can easily check if there exists some sequence $\sigma \in [\![\mathcal{T}]\!]$ violating the formula ϕ. Standard techniques to build such monitors have been developed over the years [39–41], and are beyond the scope of this paper. One only needs to know that, in this context, finite automata with basic propositions on transitions are sufficient to represent a monitor for finite traces (see e.g. [42] for a discussion on this). Note that these monitors will recognise all sequences σ violating the property, i.e. in $[\![\neg\phi]\!]$. The definition of monitors follows.

Definition 4 (Monitor). *A monitor $\mathcal{M}$ is a tuple $(M, m^0, B, \rightarrow_m)$ where:*

- *M is a finite set of states,*
- *$m^0 \in M$ is the initial state,*
- *$B \subseteq M$ is a set of final "bad" states,*
- *$\rightarrow \subseteq M \times \mathcal{P} \times M$ is a transition relation.*

Composition We have seen that the monitoring problem reduces to determine if a given trace $\mathcal{T} = (E, \lambda, \preceq)$ and monitor $\mathcal{M} = (M, m^0, B, \rightarrow_m)$ have a common accepted sequence of events; or in other words does there exist a sequence of events of E, compatible with $\preceq$ such that the execution of this sequence can lead $\mathcal{M}$ in a "bad" state.

A priori, we need to examine how the monitor reacts to every interleaving of $[\![\mathcal{T}]\!]$. A monitor reacts to an event e if, in its current state, there exists an



(a) Trace (b) Monitor

(c) Composition

Fig. 1. Example of composition

outgoing transition labeled with a proposition p such that $e \models p$. Let $\mathsf{succ}(m, e)$ denote the set of monitor states reached by triggering an event e, from a monitor state m. Formally, succ is defined as follows:

$$\mathsf{succ}(m,e) = \begin{cases} \{m\} & \text{if } \forall m \xrightarrow{p}_m m' : e \not\models p \\ \{m' \mid \exists m \xrightarrow{p}_m m' : e \models p\} & \text{otherwise} \end{cases}$$

According to this definition, when no transition is provided for an event, the monitor remains in its current state (i.e. does not block). This leads us to the following definition of composition of a trace with a monitor.

Definition 5 (Composition). *The composition of a trace $\mathcal{T}$ and a monitor $\mathcal{M}$, noted $\mathcal{T} \otimes \mathcal{M}$ is a transition system $(Q, q^0, \rightarrow)$ where:*

- $Q = 2^E \times M$ *is the set of configurations*
- $q^0 = (\emptyset, m^0)$ *is the initial configuration*
- $\rightarrow \subseteq Q \times E \times Q$ *is the transition relation defined as follows:* $\forall (s,m) \in Q, \forall e \in \mathsf{enabled}(s)$, $\forall m' \in \mathsf{succ}(m,e)$

$$(s,m) \xrightarrow{e} (s \cup \{e\}, m')$$

We note $(s,m) \overset{\sigma}{\rightsquigarrow} (s',m')$ iff $\exists (s_0,m_0),(s_1,m_1),...,(s_n,m_n)$, such that $(s,m) = (s_0,m_0), (s',m') = (s_n,m_n)$ and the path $\sigma = e_1 e_2 \cdots e_n$ with $\forall\ 0 \leq i < n\ :\ (s_i,m_i) \xrightarrow{e_i} (s_{i+1},m_{i+1})$. We also note $(s,m) \rightsquigarrow (s',m')$ if $\exists \sigma \in [\![T]\!]\ :\ (s,m) \overset{\sigma}{\rightsquigarrow} (s',m')$, and $\mathsf{reachable}(s,m) = \{m' \in M \mid \exists s'\ (s,m) \rightsquigarrow (s',m')\}$. A simple example of composition is presented in figure 1 where e.g. the vector $\langle 2,1 \rangle$ represents the cut reached after execution of 2 events in $P1$ (x:=0; y:=3) and 1 event in $P2$ (w:=4) (i.e. $\langle 2,1 \rangle = \{x := 0, y := 3, w := 4\}$)

Using this composition, the trace monitoring problem can be easily solved by checking that $\mathsf{reachable}(s,m) \cap B = \emptyset$. However, the number of configurations in

the composition can be exponential in the number of processes. In fact, we have proved in a previous work [13] that the TMP is NP-complete.

5 Symbolic composition

Our aim is to exploit the fact that the monitor is not always sensitive to all events, In order to reduce the number of interleavings to explore. Indeed, in the classical exploration, if an event e does not assign any variable appearing in a guard of an outgoing transition, we consider two cases: one where e is fired, and one where e is not. But both executions correspond to the same evolution of the monitor. Hence, it would be more efficient to remember that e *optionally* has been fired. However, such an event might become relevant in the future because its execution either induces directly, or indirectly, a monitor move. Therefore, in our approach, each configuration will separate both kinds of events: *optional* events, i.e. events that did not take part in a monitor move, and *mandatory* events, i.e. events that did take part in a monitor move. In practice, such a configuration is a tuple (t, w, m), where t and w are cuts. *Mandatory* events are contained in t, and *optional* events are contained in between t and w. One of these configuration (t, w, m) symbolically represents an entire set of *explicit* configuration $\{(s, m) \mid s \in \mathcal{C}_\mathcal{T} \wedge t \subseteq s \subseteq w\}$. Hence in the following we will refer to these as symbolic configuration.

Definition 6 (Symbolic Composition). *The symbolic composition of a trace $\mathcal{T}$ and a monitor $\mathcal{M}$, noted $\mathcal{T} \otimes_s \mathcal{M}$ is a transition system $(Q_s, q_s^0, \rightarrow_s)$ where:*

- $Q_s = 2^E \times 2^E \times M$ *is the set of symbolic configurations*
- $q_s^0 = (\emptyset, \emptyset, m^0)$ *is the initial symbolic configuration*
- $\rightarrow_s \subseteq Q_s \times E \times Q_s$ *is the transition relation defined* $\forall (t, w, m) \in Q_s$, *as follows:*

  (i) if $e \notin \mathsf{sensitive}(m) \ \wedge \ e \in \mathsf{enabled}(w)$ *, then*

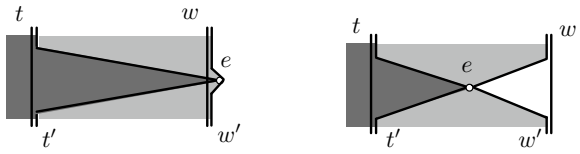
  $$(t, w, m) \xrightarrow{e}_s (t, w \cup \{e\}, m)$$

  (ii) if $e \in \mathsf{sensitive}(m) \ \wedge \ e \in \mathsf{enabled}(w) \cup [t, w]$, *then* $\forall m' \in \mathsf{succ}(m, e)$

  $$(t, w, m) \xrightarrow{e}_s (t \cup \downarrow e, (w \setminus \uparrow e) \cup \{e\}, m')$$

The relations $\overset{\sigma}{\rightsquigarrow}_s$ and $\rightsquigarrow_s$ are defined similarly to $\overset{\sigma}{\rightsquigarrow}$ and $\rightsquigarrow$ of the previous section. We note $\mathsf{reachable}_s(t, w, m) = \{m' \in M \mid (t, w, m) \rightsquigarrow (t', w', m')\}$, the set of reachable monitor states.

As illustrated in figure 2, from a symbolic configuration (t, w, m), we can fire events that were not previously examined before (Fig. 2(a)), or events that were examined before as *optional* and that have become interesting now (Fig. 2(b)). When firing an event e, we consider two cases. The first case is when e is not sensitive in m. In this case, since e becomes *optional*, it is simply added to w. On the other hand, if e is sensitive in m, it becomes *mandatory* and must be



(a) $e \in \mathsf{enabled}(w)$ (b) $e \in [t, w]$

Fig. 2. Symbolic transition $(t, w, m) \xrightarrow{e}_s (t', w', m')$ with $e \in \mathsf{sensitive}(m)$

added to t. However if e becomes mandatory, so are all its causal predecessors; so $\downarrow e$ is added to t. Moreover, we add e to w in order to keep t included in w, and all events added to w in the strict future of e must be removed from w since e changed from *optional* to *mandatory*.

Theorem 1 (Correctness of symbolic composition [13]). *The symbolic composition is correct w.r.t the classical explicit composition*

$$\mathsf{reachable}_s(\emptyset, \emptyset, m^0) = \mathsf{reachable}(\emptyset, m^0)$$

Proof. A complete proof can be found in [43].

The advantage of this method is that since non-sensitive events do not influence the monitor when being fired, they can be fired in any order, e.g. before firing sensitive events. To prove this, we introduce a covering operator on symbolic configurations, and show that it is monotonic w.r.t the symbolic composition.

Definition 7 (Covering operator $\sqsubseteq$). *A symbolic configuration (t, w, m) is covered by a symbolic configuration (t', w', m'), noted $(t, w, m) \sqsubseteq (t', w', m')$, iff*

$$(t' \subseteq t) \wedge (w \subseteq w') \wedge (m = m')$$

Theorem 2 (Monotonicity of $\sqsubseteq$). *The covering operator is monotonic w.r.t the symbolic composition.*

$$\Big((t_1, w_1, m_1) \xrightarrow{e}_s (t_1', w_1', m_1') \wedge (t_1, w_1, m_1) \sqsubseteq (t_2, w_2, m_2)\Big)$$
$$\Rightarrow$$
$$\Big(\exists (t_2', w_2', m_2') : (t_2, w_2, m_2) \xrightarrow{e}_s (t_2', w_2', m_2') \wedge (t_1', w_1', m_1') \sqsubseteq (t_2', w_2', m_2')\Big)$$

Proof. A complete proof can be found in [43].

Theorem 2 says that if a symbolic configuration (t_1, w_1, m_1) is covered by another symbolic configuration (t_2, w_2, m_2), then (t_2, w_2, m_2) can simulate (t_1, w_1, m_1), i.e. have at least the same behaviours. Moreover, it is easy to see that, for any configuration (t, w, m), and any event $e \in \mathsf{enabled}(w) \setminus \mathsf{sensitive}(m)$, $(t, w, m) \sqsubseteq (t, w \cup \{e\}, m)$. Any sensitive events that could be fired from (t, w, m), can also be fired from $(t, w \cup \{e\}, m)$. Therefore, firing non-sensitive events first is safe.

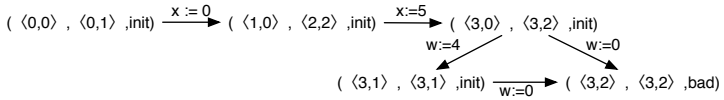


Fig. 3. Symbolic exploration

6 Online symbolic monitoring algorithm

The symbolic composition presented in the previous section can be refined into an efficient online symbolic monitoring algorithm, presented in algorithm 1. This algorithm receives and treats events one at a time, in any order. It maintains a set T of all active symbolic configurations at any given time. It also maintains two sets of events. The first set E contains all events that have already been received and treated. The second set F contains the events that have been received, but that could not be treated right away because their past was incomplete at the time. F is only used to obtain events in a topological order. Obviously, if the various processes send their events quickly, F should remain small. When a new event is received, it is added to F (line 5). Every event e from F that is ready (i.e. whose past is complete) is then treated. For that, we examine T to determine all symbolic configurations from which e can be fired (line 10). Given such a symbolic configuration (t, w, m), we first explore all non-sensitive events, as explained earlier. In practice, we add to w all non-sensitive events of $\mathsf{enabled}(w)$, and this repeatedly until a fixed point is reached (lines 14–17). From the resulting configuration, every sensitive events is fired yielding several new configurations (lines 21–22). The last part of the loop consists in cleaning T from any unnecessary symbolic configuration. First we remove any symbolic configuration from which a bad state can never be reached anymore (line 24). Note that a statical analysis of the monitor can be used to compute those "good" states beforehand. Next, any symbolic configuration that is covered by another in T is removed (line 25). Finally, outdated configurations, i.e. configurations from which no new event can be fired, are removed (line 26). In order to determine if a symbolic configuration (t, w, m) is outdated, we just check that since the cut w, at least one event has been received from every process.

7 Experimental results

In this section, we try to experimentally validate our method. We tried to compare our symbolic algorithm, as presented in the previous section, with an explicit level-by-level exploration of the lattice of cuts. This explicit algorithm was taken from [28] and adapted after discussing it with the authors [38] to make it fully online.

Traces were generated by instrumenting the code to emit relevant events (i.e. assignments). The partial order relations were obtained using vector clocks. For each example, we compared the mean number of active configurations at any given time, as well as the mean time needed for the monitoring process to react

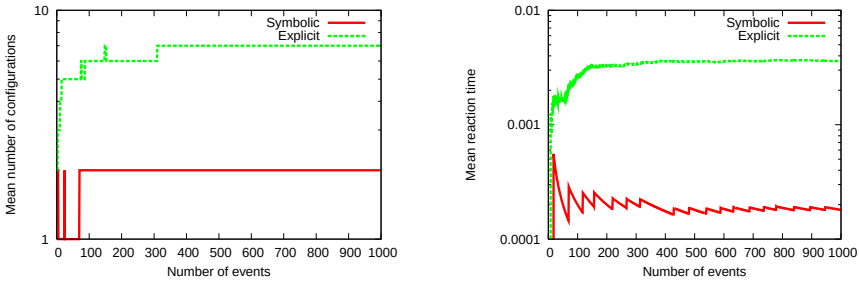
Algorithm 1: Online symbolic monitoring

```
   input: M = (M, m^0, B, →_m), n (the number of processes)
   output: ERROR if the property is violated and OK if the property is satisfied
 1 begin
 2     E ← ∅, F ← ∅
 3     T ← {(∅, ∅, m^0)}
 4     while T ≠ ∅ do
 5         F ← F ∪ {receiveEvent()}
 6         while (F ∩ enabled(E) ≠ ∅) do
 7             let e ∈ F ∩ enabled(E)
 8             F ← F \ {e}
 9             E ← E ∪ {e}
10             W ← {(t, w, m) | e ∈ enabled(w)}
11             while W ≠ ∅ do
12                 let (t, w, m) ∈ W
13                 W ← W \ {(t, w, m)}
14                 repeat
15                     x ← w
16                     w ← w ∪ (enabled(w) \ sensitive(m))
17                 until (w = x)
18                 T ← T ∪ {(t, w, m)}
19                 if (m ∈ B) then
20                     return ERROR
21                 forall e' ∈ (enabled(w) ∪ [t, w]) ∩ (sensitive(m)) do
22                     W ← W ∪ {(t ∪ ↓e', w \ ↑e' ∪ {e'}, m') | m' ∈ succ(m, e')} \ T
23             T ← T \ {(t, w, m) ∈ T | ∀m' ∈ M : (m ⇝ m') ⇒ (m' ∉ B)}
24             T ← T \ {(t, w, m) ∈ T | ∃(t', w', m') ∈ T : (t, w, m) ⊑ (t', w', m')}
25             T ← T \ {(t, w, m) | ∀1 ≤ i ≤ n ∀e ∈ w ∩ P_i ∃e' ∈ E ∩ P_i : e ≺ e' }
26     return OK
27 end
```

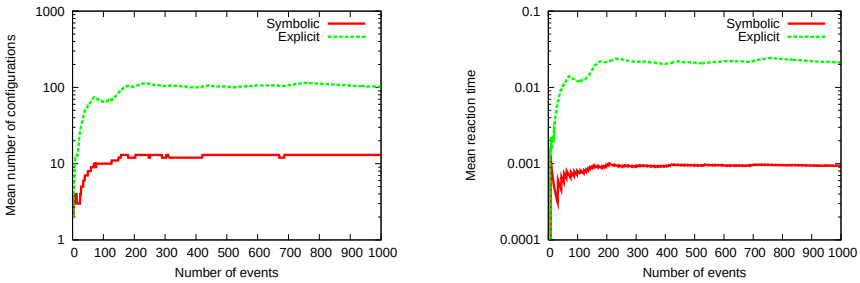
to a new event. Note that the scale on the y axis of the graphs are logarithmic. A sample of the numerical results of these experiments can be found, for the reviewers, in appendix A

The first example we considered was the *Alternating Bit* protocol with two processes, where communication is done through message passing. We used a monitor to check that no two messages with the same number (0 or 1) are successully transmitted consecutively. Figure 4(a) shows that even in such a simple example, with only two processes, the symbolic approach behaves better than the explicit one. In terms of number of configurations, this could be expected, since the method was design for that very purpose. For the reaction time, however, one could have thought that the rather heavy machinery of the symbolic composition would have considerably slowed the monitoring process. Fortunately, it appears not to be the case since the symbolic algorithm run in the average more than 10 times faster.

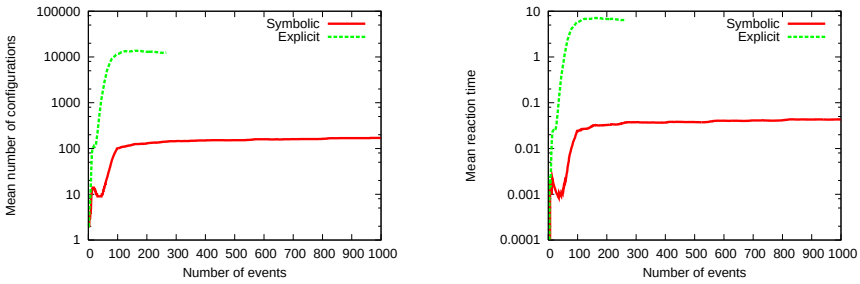
The other example we considered was the *Dining Philosophers* problem. In this example, communication between processes is done through shared variables. The monitor was used to check that no deadlock occurs, i.e. each philosopher has one fork and waits for the other. We considered 3 and 5 philosophers. To the contrary of the previous example, in the dining philosophers problem, processes are much more independent, allowing more interleavings of events. In



(a) Alternating Bit Protocol



(b) Dining Philosophers Problem (3 philosophers)



(c) Dining Philosophers Problem (5 philosophers)

Fig. 4. Experimental results

this case, depicted in figures 4(b) and 4(c), we can clearly see that the symbolic approach outperforms the explicit one. With only 5 philosophers, the explicit algorithm runs out of memory (2Gb) after 250 events. Moreover, the reaction time goes over 6 seconds, which is unacceptable when monitoring industrial controlers.

Event though the results presented in this section are only preliminary, the few examples we studied indicate that the symbolic approach presented in this paper is quite promising.

8 Further works

Our symbolic method will be integrated into our distributed controllers design environment $_{\mathrm{d}}$SL [1, 2] to allow efficient testing of real industrial distributed controllers. It will allow us to run more experiments to make its full validatation.

We will also investigate on the use of our method in different frameworks. A first candidate is the validation of Message Sequence Charts (MSC). We must study how our method can improve the efficiency of existing MSC validation methods.

As a next step, we will see how it is possible to distribute our monitors, and to extend its goal to control the system with possible recovery from failure. Indeed, the needed systematic communication between the processes and the centralized monitor to transfer the recorded events is a clear drawback of our method. From the theoretical works previously made in the domain, it is also clear that it will be impossible to completely remove the communication; but a drastic reduction could be hoped to run the distributed monitor. Notice that, our know-how in efficient automatic code distribution [1, 2] could be useful in this task.

Finally, we are also interested in the extension of our method to the model-checking of complete systems. The combined use of our method with unfolding technique developed by McMillan [33] and further refined by Esparza [34] seems a priori a promising approach.

References

1. De Wachter, B., Massart, T., Meuter, C.: dsl : An environment with automatic code distribution for industrial control systems. In: Lecture Notes in Computer Sciences. Volume 3144. Springer (2004) 132–145 (14 pages).
2. De Wachter, B., Genon, A., Massart, T., Meuter, C.: The formal design of distributed controllers with dsl and spin. Formal Aspects of Computing **17**(2) (2005) 177–200 (24 pages).
3. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. **23**(5) (1997) 279–295
4. McMillan, K.: The smv system. Technical Report CMU-CS-92-131, Carnegie Mellon University (1992)
5. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: CAV. (2002) 359–364
6. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Volume 1032 of Lecture Notes in Computer Science. Springer (1996)
7. Valmari, A.: On-the-fly verification with stubborn sets. In: CAV. (1993) 397–408
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
9. McMillan, K.L.: Symbolic model checking: an approach to the state explosion problem. Carnegie Mellon University (1992)
10. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3) (1992) 293–318

11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
12. Mattern, F.: Virtual time and global states of distributed systems. In et al., C.M., ed.: Proc. Workshop on Parallel and Distributed Algorithms, North-Holland / Elsevier (1989) 215–226
13. Genon, A., Massart, T., Meuter, C.: Monitoring distributed controllers: When an efficient ltl algorithm on sequences is needed to model-check traces. In J. Misra, T.N., ed.: FM. Lecture Notes in Computer Science, Springer (2006) to appear.
14. Mazurkiewicz, A.W.: Trace theory. In: Advances in Petri Nets. (1986) 279–324
15. Thiagarajan, P.S.: A trace based extension of linear time temporal logic. In Abramsky, S., ed.: Proceedings of the Ninth Annual IEEE Symp. on Logic in Computer Science, LICS 1994, IEEE Computer Society Press (1994) 438–447
16. Alur, R., Peled, D., Penczek, W.: Model checking of causality properties. In: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95), San Diego, California (1995) 90–100
17. Thiagarajan, P.S., Walukiewicz, I.: An expressively complete linear time temporal logic for mazurkiewicz traces. Inf. Comput. **179**(2) (2002) 230–249
18. Diekert, V., Gastin, P.: LTL is expressively complete for Mazurkiewicz traces. Journal of Computer and System Sciences **64**(2) (2002) 396–418
19. Chase, C.M., Garg, V.K.: Detection of global predicates: Techniques and their limitations. Distributed Computing **11**(4) (1998) 191–201
20. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1) (1985) 63–75
21. Charron-Bost, B., Delporte-Gallet, C., Fauconnier, H.: Local and temporal predicates in distributed systems. ACM Trans. Program. Lang. Syst. **17**(1) (1995)
22. Garg, V.K., Waldecker, B.: Detection of weak unstable predicates in distributed programs. IEEE Trans. Parallel Distrib. Syst. **5**(3) (1994) 299–307
23. Garg, V.K., Waldecker, B.: Detection of strong unstable predicates in distributed programs. IEEE Trans. Parallel Distrib. Syst. **7**(12) (1996) 1323–1333
24. Garg, V.K., Mittal, N.: On slicing a distributed computation. In: ICDCS. (2001) 322–329
25. Mittal, N., Garg, V.K.: Computation slicing: Techniques and theory. In: DISC. (2001) 78–92
26. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: OPODIS. (2003) 171–183
27. Sen, K., Rosu, G., Agha, G.: Online efficient predictive safety analysis of multithreaded programs. In: TACAS. (2004) 123–138
28. Sen, K., Rosu, G., Agha, G.: Detecting errors in multithreaded programs by generalized predictive analysis of executions. In: FMOODS. (2005) 211–226
29. Chen, F., D'Amorim, M., Roşu, G.: Checking and correcting behaviors of java programs at runtime with java-mop. In: Workshop on Runtime Verification (RV'05). ENTCS (2005)
30. Debouk, R., Lafortune, S., Teneketzis, D.: Coordinated decentralized protocols for failure diagnosis of discrete event systems. Discrete Event Dynamic Systems: Theory and Applications **10**(1) (2000) 33–86
31. Genc, S., Lafortune, S.: Distributed diagnosis of discrete-event systems using petri nets. In van der Aalst, W.M.P., Best, E., eds.: ICATPN. Volume 2679 of Lecture Notes in Computer Science., Springer (2003) 316–336
32. Benveniste, A., Fabre, E., S.Haar, Jard, C.: Diagnosis of asynchronous discrete event systems, a net unfolding approach. IEEE Transactions on Automatic Control **48**(5) (2003) 714–727

33. McMillan, K.L.: A technique of state space search based on unfolding. Formal Methods in System Design **6**(1) (1995) 45–65
34. Esparza, J., Römer, S., Vogler, W.: An improvement of mcmillan's unfolding algorithm. (1996) 87–106
35. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Inf. Process. Lett. **90**(1) (2004) 21–28
36. Puri, A., Tripakis, S., Varaiya, P.: Problems and examples of decentralized observation and control for discrete event systems. In B. Caillaud, P. Darondeau, L.L., Xie, X., eds.: Synthesis and Control of Discrete Event Systems. Kluwer Academic Publishers (2002) 37–56
37. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Proceedings of 26th International Conference on Software Engineering (ICSE'04), Edinburgh, UK, IEEE (2004) 418–427
38. Chen, F.: Private communication. (2006)
39. Jard, C., Jéron, T.: On-line model-checking for finite linear temporal logic specifications. **407** (1989) 275–285
40. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design **19**(3) (2001) 291–314
41. Geilen, M.: On the construction of monitors for temporal logic properties. Electr. Notes Theor. Comput. Sci. **55**(2) (2001)
42. Leuschel, M., Massart, T., Currie, A.: How to make fdr spin : Ltl model checking of csp by refinement. In: Lecture Notes in Computer Sciences. Volume 2021. Springer (2001) 99–118 (20 pages).
43. Genon, A., Massart, T., Meuter, C.: Monitoring distributed controllers : When an efficient ltl algorithm on sequences is needed to model-check traces. Technical Report 2006-59, CFV - Université Libre de Bruxelles (2006)

A Experimental data (for reviewers)

		#Conf.		Reaction-time (sec.)	
Experiment	#Events	Symbolic	Explicit	Symbolic	Explicit
ABP	100	2	6	0.0002	0.0026
	250	2	6	0.0002	0.003254
	500	2	7	0.00018	0.00356
	1000	2	7	0.00018	0.00359
Philosophers 3	100	10	65	0.000818	0.0121
	250	12	111	0.000920	0.02328
	500	13	102	0.000960	0.02112
	1000	13	102	0.000940	0.02117
Philosophers 5	100	100	11639	0.0238	5.8324
	250	138	12383	0.0352	6.43256
	500	152	-	0.0378	-
	1000	170	-	0.0433	-

Results for the explicit and symbolic monitoring: mean number of configurations and execution time per received event after a sequence of n events.