Monitoring Distributed Controllers: When an Efficient LTL Algorithm on Sequences is Needed to Model-Check Traces

Alexandre Genon, Thierry Massart, and Cédric Meuter[⋆]

Université Libre de Bruxelles[⋆⋆]

Abstract. It is well known that through code instrumentation, a distributed system's finite execution can generate a finite trace as a partially ordered set of events. We motivate the need to use LTL model-checking on sequences and not on traces as defined by Diekert and Gastin, to validate distributed control systems executions, abstracted by such traces, and present an efficient symbolic algorithm to do the job. It uses the standard method proposed by Vardi and Wolper, which from the LTL formula, builds a monitor that accepts all the bad sequences. We show that, given a monitor and a trace, the problem to check that both the monitor and the trace have a common sequence is NP-complete in the number of concurrent processes. Our method explores the possible configurations symbolically, since 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 exploration method, with or without partial order reduction (which, in practice, does not work well here).

Keywords: testing of asynchronous distributed systems, monitor, global properties, model checking of traces

1 Introduction

A distributed control system is a set of distributed hardware equipments such as small computers or Programmable Logic Controllers (PLCs), which run concurrent processes, communicating asynchronously through some network. The design and implementation of such a distributed reactive system is a non-trivial task. Validation and debugging techniques can be used during the design and the implementation to help the developer in his work [DMM04,DGMM05]. *Verification* tools (e.g. [Hol97,McM92b,CCG$^+$02]) can be used to validate a model. Unfortunately in practice, the system's implementation code contains thousands of lines and dozens of variables. The *state-explosion problem* generally prevents

⋆ {tmassart,cmeuter}@ulb.ac.be

⋆⋆ Boulevard du Triomphe - CP-212, 1050 Bruxelles, Begium - Tel:+32 2 650.5603 - Fax:+32 2 650.5609 the designer from its exhaustive verification even with efficient exploration techniques such as partial order reduction [God96,Val93] or symbolic model checking [CGP99,McM92a,Bry92].

The designer generally falls back to *testing*, which cannot guarantee that a system is completely bug-free, but if achieved on a large number of test-cases (e.g. covering all the system's functionalities), can give a *reasonable* confidence that the system is correct. For that purpose, the implementation is generally instrumented to record relevant events. A special process, called *monitor*, records the system's events and must then check that the observed execution satisfies the desired properties. This monitoring can either be done offline, i.e. after the complete trace is recorded, or online, at runtime. Notice that this monitoring technique can also be used to validate runs of a system's *model*, if too complex to be exhaustively verified. Hence, both at the design and implementation level, it is an important activity where efficient methods must be provided.

For distributed asynchronous systems [Lyn96], a run is generally not seen as a totally ordered sequence of events, but as a partially ordered set where unordered events may have occurred in any order. In a simple approach, the monitor just assumes that the events happened in the order they are received, and check that the property is satisfied. In a predictive approach [SRA04], the monitor must check that every compatible total order of events satisfies the property. The causal *partial order* between the fired events can be obtained through correct code instrumentation using, e.g. vector clocks [Lam78,Mat89]. The collected information of an execution is therefore abstracted as a *trace*, i.e. a partially ordered set of events where two consecutive events of the same site are temporally ordered and where communications (e.g. message transfers or shared variable manipulations) impose an order between some distributed events.

An important point to note is that even if the control is distributed and provides a partially ordered trace, the *exact sequence the control actions are taken is generally crucial*. One can for instance think to a controlled system where a valve A must be closed before another valve B can be opened and where each valve is controlled by another PLC; the *controlled environment* is therefore seen as centralized. *Testing* that an execution satisfies a *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*. Therefore, we will see that our traces can not be seen as Mazurkiewicz traces [Maz86] where the order of independent events is meaningless.

Unfortunately, even if the monitor is already built, this problem is hard and in practice, the number of compatible sequential executions may be exponential in the number of concurrent processes. Therefore, in the same spirit as what is done with partial order reduction techniques, which try to minimize the exploration of execution interleavings as much as possible, we investigate here an efficient method to practically reduce the verification time. Moreover our proposed method is symbolic since it handles sets of configurations. We show in practice that our method is very efficient in execution time, compared to the standard exploration method with or without partial order reduction.

This paper is organized as follows. In section 2, we detail related proposals and explain why the problem needs model-checking on sequences and not on traces. In section 3, we introduce our model for traces and monitors, formalize the trace monitoring problem and show that this problem is hard even with an already built monitor. In section 4, we present our symbolic method and show its correctness, and in section 5, we show how this method can be refined into a symbolic exploration algorithm. Next, in section 6, we present our experimental results of various examples. Finally, further works are given in section 7.

2 Related Works and motivation

In the literature, papers on *global predicate detection* and *trace model-checking* have generally a common starting point since the *system* to verify is composed of various concurrent processes synchronized by some mean. This system is modeled, possibly after some code instrumentation (using e.g. [Lam78,Mat89]), as a *trace*, i.e. a set of temporally partially ordered events.

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. Numerous works have been done on the detection of global predicates. Garg and Chase showed in [CG98] that this problem is NP-complete for an arbitrary predicate, even when there is no interprocess communication. Chandy and Lamport [CL85], present a technique for stable predicates, i.e. predicates that never turn false once they become true. In [CDF95], Charron-Bost *et al* present an algorithm for observer independent predicates. In [GW94,GW96], Garg and Waldecker give polynomial procedures for *conjunctive* predicates, i.e. predicates that are conjunctions of local predicates. In [CG98], Chase and Garg introduce the classes of *linear* and *semi-linear* predicates and give an efficient procedure to solve the predicate detection problem for these classes of systems. In [GM01], Garg and Mittal introduce the notion of *regular* predicates, a subset of the linear predicates, for which they present a procedure that solves the predicate detection problem in polynomial time. This procedure makes use of *computation slicing*, that is, computing all cuts compatible with a given execution satisfying a given predicate. They present an efficient procedure for computing such slices. Computation slicing on regular predicates is examined in details in [MG01]. In [SG03], 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 [SRA04] 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. In [SVAR04] K. Sen et al. define the logic PT-DTL which is a variant of past time linear temporal logic, suitable for efficient distributed model-checking on execution traces. 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 [SRA04]. We investigate here on the possibility to define a method, efficient in practice to be able to model-check any LTL formula. Therefore, we do not limit the exploration as in [SRA04], but prefer to increase its efficiency with a symbolic method.

Trace model checking has been studied through the definition of several linear temporal logics for Mazurkiewicz traces. A Mazurkiewicz trace [Maz86], 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 [Thi94] and Alur, Peled and Penczek on TLC [APP95]. Global trace logics include, among others, LTrL [TW02] proposed by Thiagarajan and Walukiewicz, and LTL on traces [DG02] defined by Diekert and Gastin.

However, in our 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. Therefore, 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*. Therefore, since we do not have a priori the independence relation, these trace logics are not adapted to model-check our runs.

3 Trace monitoring problem

In this section, we first introduce our framework with the notions of *finite trace* which models a run of a concurrent system, and *monitor* which is an automata representation of the formula to check. Then, we formalize the trace monitoring problem and prove its NP-completeness in the number of concurrent processes.

Trace Our runs are obtained by concurrent processes, each executing a finite sequence of variables assignments. Moreover, due to inter-process communications, other causal relations 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. Formally:

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,*
- *$\lambda : E \mapsto Var \times \mathbb{N}$ 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 simple variable x and value v of the corresponding assignment.*
- *$\preceq \subseteq E \times E$ is a partial order relation on E*

In the following, we will use the following notations: $\downarrow e$ denotes $\{e' \mid e' \preceq e\}$ and $\uparrow e$ denotes $\{e' \mid e \preceq e'\}$ (the reflexo-transitive closure of resp. causal predecessors and successors). Moreover, for any set S of events, $\uparrow S = \cup_{e \in S} \uparrow e$ and $\downarrow S = \cup_{e \in S} \downarrow e$. 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 already executed 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 mentioned earlier, even if our systems are finite traces, their particular nature, i.e., the fact that they come from a distributed controller of a global environment which can be seen as centralized, induces that their semantics is defined classically by the sets of (finite) sequences of events they can do.

Definition 2 (Semantics of a trace). *For a set of variables Var, and a trace $\mathcal{T} = (E, \lambda, \preceq)$ defined with these variables, the semantics $[\![\mathcal{T}]\!]$ is defined as the set of sequences of execution $\mathcal{T}$ can have. Formally:*

$$[\![\mathcal{T}]\!] = \left\{ \sigma = e_1, e_2, ..., e_{|E|} \mid \forall 1 \leq i, j \leq |E| : \begin{array}{c} (e_i \in E) \wedge (e_i \preceq e_j) \Rightarrow (i \leq j) \wedge \\ (i \neq j) \Rightarrow (e_i \neq e_j) \end{array} \right\}$$

Remark: if needed we can easily define the value of the variables at some point in the execution. At the beginning of the execution, we can assume all variables to be equal to 0. However, with our model, we cannot in general, talk about the value of a variable x in some cut C; this value can depend on the particular path σ taken to reach C.

Monitor Now that we have defined the model $\mathcal{T}$ of a distributed system, we need to define how a property can be expressed on $\mathcal{T}$.

Since events in a trace are single assignment, we naturally first define basic formulae 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 formula* uses only one variable of the trace. Example of such basic formulae are $(x = 3)$ or $((0 < 2 * x) \wedge (2 * x < 5))$. We denote by $\mathcal{F}$ the set of such basic formulae and by $\mathsf{var}(\phi)$, the variable appearing in a basic formula ϕ. For a given basic formula ϕ, and an event e which executes the assignment $x := v$, we naturally define :

Definition 3 (Formula triggering). *An event e triggers a formula ϕ, if e assigns the variable appearing in ϕ and if ϕ evaluates to true after the assignment. Formally, if $\phi[x \leftarrow v]$ denotes the formula ϕ where the variable x is substituted by v, then we have:*

$$(\phi \models e) \Leftrightarrow ((\mathsf{var}(e) = \mathsf{var}(\phi)) \wedge (\phi[\mathsf{var}(e) \leftarrow \mathsf{val}(e)] = \top))$$

Those basic formulae can be used as propositions to build more complex temporal constraints, using LTL.

A particular care must be taken to the fact that it will be checked on finite sequences. This can be done, as explained e.g. in [LMC01] by an obvious translation of any LTL formula into an "LTL with Δ actions" (where Δ is not in the initial alphabet). Semantically the finite sequences are extended by an infinite sequence of Δ actions, to mark the deadlock. For example, intuitively a system S should satisfy $\neg a$ iff S can not perform a a as next action. Hence S may either perform only actions b different from a or it may deadlock. Similarly, if $\bigcirc$ denotes the *next* operator in LTL, a system which satisfies $\neg \bigcirc a$ can either deadlock immediately or perform some visible action and then satisfy $\neg a$. To capture the intuition, any formula $\bigcirc\phi$ is first translated into $(\neg\Delta \ \wedge \ \bigcirc\phi)$ and $\neg \bigcirc \phi$ into $(\Delta \ \vee \ \bigcirc\neg\phi)$.

Then, the classical procedure defined by Vardi and Wolper [VW86] to build from a LTL formula a corresponding *(Büchi) automaton* $\mathcal{B}$ able to do all the sequences of $[\![\neg\phi]\!]$ can be used to build our monitor (seen as a standard non deterministic finite automaton). The construction is restricted to our systems where only one variable is modified at each event. As explained in [LMC01], the Δ-transitions can be removed from the *monitor* obtained and a finite automaton is provided where transitions are labeled by basic formulae and with a standard and not the Büchi acceptance condition. Note that the size of the obtained monitor may be exponential in the size of the corresponding LTL formula [VW86]; but generally, since in practice the size of the formula is small, it is not a problem.

We will show that our main contribution in this paper, is an algorithm which outperforms classical methods to compose $\mathcal{B}$ and $\mathcal{T}$ and verify that $[\![\mathcal{T}]\!] \cap [\![\neg\phi]\!] = \emptyset$, i.e. check that no sequence of the system has the property ϕ.

In the following, we simply define our monitors as any non deterministic finite automata with basic formulae on transitions. The formal definition of a monitor 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,*
- *$\rightarrow \subseteq M \times \mathcal{F} \times M$ is a transition relation.*

The monitoring problem 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 total order on E, compatible with $\preceq$ such that, if the events of E are executed in that order, $\mathcal{M}$ can reach to a "bad" state. A priori, we need to examine how the monitor reacts to every interleaving of the events in E compatible with the partial order $\preceq$. A monitor reacts to an event e if, in its current state, there exists an outgoing transition labeled with a guard ϕ such that e triggers ϕ. $\mathsf{next}(e, m)$ denotes the set of monitor states reached by triggering an event e, from a monitor state m. Formally, next is defined as follows:

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

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:*

- *$Q \subseteq 2^E \times M$ 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: $\forall (s, m) \in Q, \forall e \in$ $\mathsf{enabled}(s)$, $\forall m' \in \mathsf{next}(m, e)$*

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

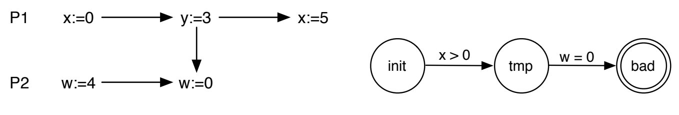
We note $(s, m) \overset{\rho}{\rightsquigarrow} (s', m')$ iff $\exists (s_0, m_0), (s_1, m_1), ..., (s_n, m_n)$, such that $(s, m) = (s_0, m_0), (s', m') = (s_n, m_n)$ and the path $\rho = e_1 e_2 \cdots e_n$ with $\forall\ 0 \leq i < n\ :\ (s_i, m_i) \xrightarrow{e_i} (s_{i+1}, m_{i+1})$. We also note $(s, m) \rightsquigarrow (s', m')$ if $\exists \rho\ :\ (s, m) \overset{\rho}{\rightsquigarrow} (s', m')$, and $\mathsf{reachable}(s, m) = \{m' \in M \mid (s, m) \rightsquigarrow (E, m')\}$. An simple example of composition is presented in figure 1 where e.g. the vactor $[2, 1]$ represents the cut reached after execution of 2 events in $P1$ (x:=0; y:=3) and 1 event in $P2$ (w:=4).

Using these notations, we can reformulate the monitoring problem.

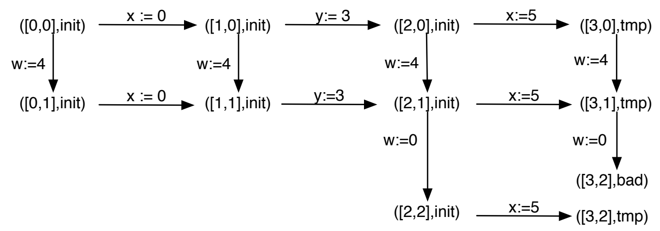
Definition 6 (Trace monitoring problem (TMP)). *Given a trace $\mathcal{T} = (E, \lambda, \preceq)$ and a monitor $\mathcal{M} = (M, m^0, B, \rightarrow_m)$ the trace monitoring problem (TMP) is to check whether* $\mathsf{reachable}(\emptyset, m_0) \cap B = \emptyset$

Remember that, by the definition of $\mathsf{reachable}$, we ask here to execute the complete trace before checking that the reached state is in B.

NP**-completeness** We now show that the monitoring problem is NP-complete in the number of concurrent processes in the trace even if the formula only uses boolean variables and where every formula on transitions is of the form $x = v$. This result is *a priori* not completely obvious since we consider restricted monitors that use single variables predicates on each transition and it is known ([GW96,GW94]) that checking conjunction of local predicates is polynomial in the size of the conjunction. We present a polynomial time reduction from 3-SAT to our problem.



(a) Trace (b) Monitor



(c) Composition

Fig. 1. Example of composition

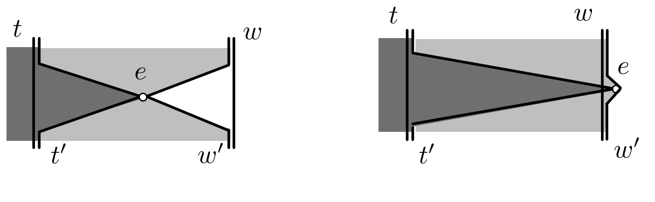
Theorem 1 (NP-completeness of TMP). *[GMM06] The trace monitoring problem is* NP*-complete.*

Proof sketch. It is easy to see that the TMP is in NP*. Indeed, one could use a non deterministic algorithm to guess an execution (of size $|E|$) and check, in polynomial time, if the corresponding total order is compatible with $\mathcal{T}_\phi$ and if this execution leads to a state in B. For NP-hardness, we reduce from 3-*SAT*. The main idea is to use a monitor to model a 3-*SAT *formula and the trace to model all possible valuations of its propositions. The only technicality resides in the fact that the valuations must not contain any pair of complementary literals. This is accomplished by properly choosing the partial order.*

Note on partial order reduction Using partial order reduction [God96] to improve the explicit exploration will not work to solve the TMP. This is because in the trace monitoring problem, the monitor expresses constraints on all, or most of the events of the trace. Therefore, no events is seen as "invisible"; and the partial order reduction brings no improvement. This was another motivation to find an effective method for the TMP. Our method is presented in the following section.

4 Symbolic composition

The main idea behind the symbolic exploration is to exploit the fact that the monitor is not always sensitive to all events. 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 consider only one execution, where e has been *optionally* fired. However, we must remember these events, because they might



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

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

become relevant in the future, i.e. they could become *mandatory* in the future. Therefore, in our approach, each symbolic configuration will separate both kinds of events: *optional* events, i.e. events that did not produce a monitor move and do not change its state if they are not taken, and *mandatory* events, i.e. events that did produce, directly or indirectly, a monitor move. In practice, a symbolic 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 $w \backslash t$ (denoted $[t, w]$ in the following). Such a symbolic configuration represents an entire set of explicit configurations $\{(s, m) \mid s \in \mathcal{C}_\mathcal{T} \wedge t \subseteq s \subseteq w\}$.

In order to define the symbolic composition based on this idea, we first need to introduce some notations. We define $\mathsf{sensitive}(m) = \{e \in E \mid \exists m \xrightarrow{\phi}_m m' : e \models \phi\}$, the set of all events that will trigger a monitor move when it is in state m.

Definition 7 (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:*

- *$Q_s \subseteq 2^E \times 2^E \times M$ 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 $\forall m' \in \mathsf{next}(m, e)$

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

We note $(t, w, m) \overset{\rho}{\rightsquigarrow}_s (t', w', m')$ iff $\exists (t_0, w_0, m_0), ..., (t_n, w_n, m_n)$, such that $(t, w, m) = (t_0, w_0, m_0), (t', w', m') = (t_n, w_n, m_n)$ and such that the path $\rho = e_1 e_2 \cdots e_n$ with $\forall\ 0 \leq i < n\ :\ (t_i, w_i, m_i) \xrightarrow{e_i}_s (t_{i+1}, w_{i+1}, m_{i+1})$. We also note $(t, w, m) \rightsquigarrow_s (t', w', m')$ if $\exists \rho\ :\ (t, w, m) \overset{\rho}{\rightsquigarrow}_s (t', w', m')$, and $\mathsf{reachable}_s(t, w, m) = \{m' \in M \mid (t, w, m) \rightsquigarrow (t', E, m')\}$, the set of monitor states, reachable at the end of a trace's run.

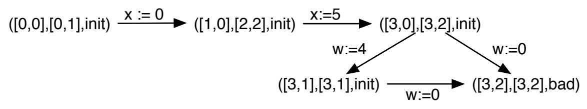


Fig. 3. Symbolic exploration

From a symbolic configuration (t, w, m), we can fire events that were not previously examined before (events in $\mathsf{enabled}(w)$), or events that were examined before as *optional* and that allows now to change the current minitor state (event in $[t, w] \cap \mathsf{sensitive}(m)$). 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 added to t *together with all its causal predecessors* ($\downarrow e$). 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*.

We now need to prove that this symbolic composition is correct w.r.t the classical explicit exploration.

Theorem 2 (Correctness of symbolic composition). *[GMM06] 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 sketch. First prove that the symbolic composition is consistent, i.e. that $\big((\emptyset, \emptyset, m^0) \rightsquigarrow_s (t, w, m)\big) \Rightarrow ((\downarrow t = t) \wedge (\downarrow w = w) \wedge (t \subseteq w))$. *Then, we prove that it is sound, i.e. that* $\big((\emptyset, \emptyset, m^0) \rightsquigarrow_s (t, w, m)\big) \Rightarrow \big(\forall s : (t \subseteq s \subseteq w) \wedge (\downarrow s = s) \Rightarrow (\emptyset, m^0) \rightsquigarrow (s, m)\big)$. *Finally, we prove that the symbolic composition is complete, i.e.* $\big((\emptyset, m^0) \rightsquigarrow (s, m)\big) \Rightarrow \big(\exists t, w : (t \subseteq s \subseteq w) \wedge (\emptyset, \emptyset, m^0) \rightsquigarrow_s (t, w, m)\big)$

5 Symbolic exploration

Taking advantage of the symbolic composition presented in the previous section, we propose our method, given in algorithm 1, which can efficiently solve the TMP, i.e. compute if $\mathsf{reachable}_s(\emptyset, \emptyset, m^0) \cap B = \emptyset$. The idea behind this algorithm is simple. Given a symbolic configuration (t, w, m), we first explore all non-sensitive events. Since these events do not influence the monitor when fired, the order in which they are fired is not important. Therefore, we can just consider them in any order. In practice, we add to w all non-sensitive event of $\mathsf{enabled}(w)$, and this repeatedly until a stabilization (lines 7–10). Afterwards, from there, we fire all sensitive events yielding several new symbolic configurations (lines 13–14). The sets W and T contain the symbolic configuration resp. remaining to handle and already handled. The resulting state space for the monitor and trace presented in figure 1 is presented in figure 3

Algorithm 1: Symbolic exploration

```
   input  : $\mathcal{T} = (E, \lambda, \preceq), \mathcal{M} = (M, m^0, B, \rightarrow_m)$
   output: $\mathsf{reachable}_s(\emptyset, \emptyset, m^0) \cap B \neq \emptyset$
1  begin
2      $T \leftarrow \emptyset,\ W \leftarrow \{(\emptyset, \emptyset, m^0)\}$
3      while $W \neq \emptyset$ do
4          let $(t, w, m) \in W$
5          $W \leftarrow W \setminus \{(t, w, m)\}$
6          $T \leftarrow T \cup (t, w, m)$
7          repeat
8              $x \leftarrow w$
9              $w \leftarrow w \cup \{e \in \mathsf{enabled}(w) \mid e \notin \mathsf{sensitive}(m)\}$
10         until $(w = x)$
11         if $(w = E) \wedge (m \in B)$ then
12             return false
13         forall $(t', w', m') : (t, w, m) \xrightarrow{e}_s (t', w', m') \wedge (t', w', m') \notin T$ do
14             $W \leftarrow W \cup \{(t', w', m')\}$
15     return true
16 end
```

In order to prove the correctness of this algorithm, we need to prove that firing non-sensitive events first is sufficient to detect if $\mathsf{reachable}_s(\emptyset, \emptyset, m^0)_s \cap B = \emptyset$ is not empty. For this, we introduce a covering operator.

Definition 8 (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*

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

Intuitively, (t', w', m') covers (t, w, m) if (t', w', m') represents more explicit configurations than (t, w, m).

Lemma 1 (Monotonicity of $\sqsubseteq$). *[GMM06] The covering operator is monotonic w.r.t the symbolic composition.*

$$\Big((t_1, w_1, m_1) \xrightarrow{e}_s (t'_1, w'_1, m'_1) \wedge (t_1, w_1, m_1) \sqsubseteq (t_2, w_2, m_2)\Big)$$
$$\Rightarrow$$
$$\Big(\exists (t'_2, w'_2, m'_2) : (t_2, w_2, m_2) \xrightarrow{e}_s (t'_2, w'_2, m'_2) \wedge (t'_1, w'_1, m'_1) \sqsubseteq (t'_2, w'_2, m'_2)\Big)$$

Theorem 3 (Correctness of algorithm 1). *Given a trace $\mathcal{T} = (E, \lambda, \preceq)$ and a monitor $\mathcal{M} = (M, m^0, B, \rightarrow_m)$, algorithm 1 terminates and returns true* iff $\mathsf{reachable}_s(\emptyset, \emptyset, m^0) \cap B \neq \emptyset$.

Proof. First, we can see that E and M are finite, so is the set of possible symbolic configurations. Therefore, termination is guaranteed since we do not explore a symbolic configuration more than once. Moreover, theorem 2 ensures soundness, since only valid symbolic transition are taken. However, completeness is not that trivial, because the algorithm explore only a subset of all symbolic configurations. This comes from the fact that non-sensitive events are always fired first. However,

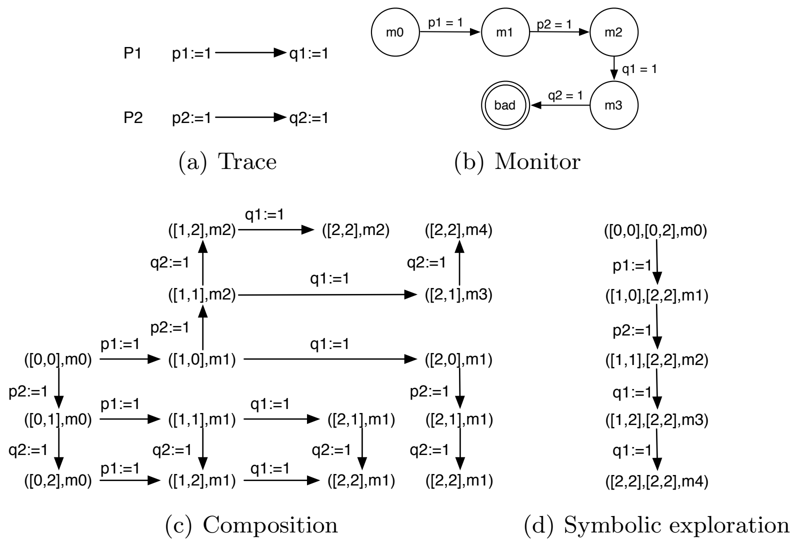


Fig. 4. Example of where algorithm 1 leads to an exponential gain

if c denotes the configuration after the loop at lines 7–10, then any configuration c' computed during the loop is covered by c. Therefore, lemma 1 guarantees that any configuration reachable from c' is covered by a configuration reachable from c. Thus, only exploring c is sufficient.

Note that using lemma 1, we could improve algorithm 1 by replacing the test $(t', w', m') \in T$, in line 13, by $(t', w', m') \sqsubseteq T$, i.e does there exist a configuration in T that covers (t', w', m'). An efficient data structure which handles set of tuples-intervals (see [DRV04] and its variant [Gan02]) can be used to perform efficiently these tests. The same efficient data structure can be used to represent W; then the union in line 14 adds a new configuration only if it is not already covered by W.

Possible exponential improvement In practice, as shown in the next section, our symbolic method, given in algorithm 1, allows generally to reduce significantly the explored state space and the verification time. We show here the simple example given in figure 4 which provides a state space which is linear with our symbolic method while exponential in the explicit method. Note that this example can be extended to k processes. Hence, by using algorithm 1, we benefit from an exponential improvement (w.r.t. k) for the state space size. For instance for 8 processes, 17 states are explored by the symbolic algorithm against 8020 for the explicit one.

6 Experimental results

In this section, we experimentally validate our method. We compared on randomly generated traces, both in time and in number configurations, our symbolic algorithm with a straightforward explicit trace exploration. Moreover, we compared our testing method with a complete model-checking using the tool *spin* [Hol97], with partial order reduction.

We conducted experiments on several examples (seen as distributed controllers). For each example, we examined a correct model and a faulty model, where a bug was intentionally introduced. Traces were generated by instrumenting the code to emit relevant events (i.e. assignments). The partial order relations were obtained using vector clocks.

Table 1 presents the results of these experiments. For each experiment, the first four columns respectively present the model, the number of processes, the number of events in the trace and the property. Next, for both the explicit, and symbolic method, columns 5 to 10 show if an error was found or not, the time needed for exploration and the number of configurations used. The last column present the times needed for the complete model-checking with partial order reduction. A "-" in the table indicates that no result could be obtained because the process ran out of memory[1]

The first example we considered was the *Peterson* mutual exclusion protocol with two processes, where communication is done through shared variables. We used a monitor to check mutual exclusion (a safety property).

The second model we considered was the *Alternating-bit protocol* between two process, i.e. a sender and a receiver. This time the communication is achieved using asynchronous channel. We used a monitor to check that every message sent was correctly received.

On those two examples, we can see that the symbolic exploration works well in practice, compared to the explicit exploration method, both with safety and liveness property. It is worth noting that, in the faulty version of both models, the error was detected rapidly. However, on those two examples, the complete *spin* model checking can be done very efficiently. This should be expected since the model is relatively small.

The last example we considered was the *Dining Philosopher* problem. The monitor was used to check that when the first philosopher takes his left fork, then his left neighbor cannot eat until he has finished to eat. Note that this property cannot be expressed in RCTL+ of [SG03] because it involves an *until* operator. We considered 3, 5 and 10 philosophers. On this example, we can see that using the symbolic method allows to handle a larger number of processes. Indeed, when dealing with 10 philosophers, the explicit exploration fails to terminate, whereas the symbolic method still works. Moreover, in the faulty model, with 10 philosophers, the complete *spin* model checking fails to terminate, whereas the symbolic exploration still detects the error that was introduced.

[1] explorations and model-checking were limited to 1GB of memory

Experiment				Explicit			Symbolic			*Spin*
Model	Processes	Events	Property	Error	Time	Conf.	Error	Time	Conf.	Error
Peterson	2	10000	Mutex	NO	1.39s	21551	NO	0.35s	4001	0.06s
	2	100000	Mutex	NO	16.88s	215544	NO	3.45s	40001	0.06s
	2	1000000	Mutex	-	-	-	NO	34.75s	400002	0.06s
Peterson Faulty	2	10000	Mutex	YES	1.11s	21384	YES	0.01s	4	0.05s
	2	100000	Mutex	YES	15.95s	214727	YES	0.05s	4	0.05s
	2	1000000	Mutex	-	-	-	YES	0.53s	4	0.05s
ABProtocol	2	10000	Received	NO	2.17s	31185	NO	0.42s	4654	0.15s
	2	100000	Received	NO	31.08s	316414	NO	4.25s	46684	0.15s
	2	1000000	Received	-	-	-	NO	43.09s	466887	0.15s
ABProtocol Faulty	2	10000	Received	YES	2.06s	31495	YES	0.01s	5	0.13s
	2	100000	Received	YES	29.70s	315808	YES	0.06s	5	0.13s
	2	1000000	Received	-	-	-	YES	0.53s	5	0.13s
Philosopher	3	100	Fork	NO	1.03s	6190	NO	0.05s	299	0.40s
	5	100	Fork	NO	87.02s	60727	NO	0.21s	2875	12.01s
	10	100	Fork	-	-	-	NO	1.52s	26791	-
Philosopher Faulty	3	100	Fork	YES	0.09s	1187	YES	0.01s	63	0.38s
	5	100	Fork	YES	78.72s	55982	YES	0.01s	78	11.01s
	10	100	Fork	-	-	-	YES	0.01s	55	-

Table 1. Experimental results ("-" means that the execution ran out of memory)

7 Future works

Our symbolic method will be integrated into our distributed controllers design environment $_{\mathrm{d}}$SL [DMM04,DGMM05] to allow efficient testing of real industrial distributed controllers. For this purpose, our method should be extended to online monitoring, and further developed to handle more complex formulae. A comparison between our tool and other existing tools could then be done both at the expressivity and performance levels.

We also intend to investigate 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. Moreover, we would like to explore the possibility of integrating other techniques such as computation slicing [MG05], in order to gain in time an space during the validation.

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 [McM95] and further refined by Esparza [ERV96] seems a priori a promising approach.

Acknowledgments We would like thank the anonymous reviewers, whose comments allowed us to significantly improve the quality of this paper.

References

[APP95] Rajeev Alur, Doron Peled, and Wojciech Penczek. Model checking of causality properties. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 90–100, San Diego, California, 1995.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[CCG$^+$02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, pages 359–364, 2002.

[CDF95] Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. *ACM Trans. Program. Lang. Syst.*, 17(1), 1995.

[CG98] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, 1999.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[DG02] Volker Diekert and Paul Gastin. LTL is expressively complete for Mazurkiewicz traces. *Journal of Computer and System Sciences*, 64(2):396–418, March 2002.

[DGMM05] Bram De Wachter, Alexandre Genon, Thierry Massart, and Cédric Meuter. The formal design of distributed controllers with dsl and spin. *Formal Aspects of Computing*, 17(2):177–200, 2005. (24 pages).

[DMM04] Bram De Wachter, Thierry Massart, and Cédric Meuter. dsl : An environment with automatic code distribution for industrial control systems. In *Lecture Notes in Computer Sciences*, volume 3144, pages 132–145. Springer, 2004. (14 pages).

[DRV04] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Covering sharing trees: a compact data structure for parameterized verification. *STTT*, 5(2-3):268–297, 2004.

[ERV96] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. In *TACAS*, pages 87–106, 1996.

[Gan02] Pierre Ganty. Algorithmes et structures de données efficaces pour la manipulation de contraintes sur les intervalles. Master's thesis, Université Libre de Bruxelles, 2002.

[GM01] Vijay K. Garg and Neeraj Mittal. On slicing a distributed computation. In *ICDCS*, pages 322–329, 2001.

[GMM06] Alexandre Genon, Thierry Massart, and Cédric Meuter. 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.

[God96] P. Godefroid. *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.

[GW94] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(3):299–307, 1994.

[GW96] Vijay K. Garg and Brian Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 7(12):1323–1333, 1996.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LMC01] M. Leuschel, T. Massart, and A. Currie. How to make fdr spin : Ltl model checking of csp by refinement. In *Lecture Notes in Computer Sciences*, volume 2021, pages 99–118. Springer, 2001. (20 pages).

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, pages 279–324, 1986.

[McM92a] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Carnegie Mellon University, 1992.

[McM92b] K.L. McMillan. The smv system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[McM95] Kenneth L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[MG01] Neeraj Mittal and Vijay K. Garg. Computation slicing: Techniques and theory. In *DISC*, pages 78–92, 2001.

[MG05] Neeraj Mittal and Vijay K. Garg. Techniques and applications of computation slicing. *Distributed Computing*, 17(3):251–277, 2005.

[SG03] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *OPODIS*, pages 171–183, 2003.

[SRA04] Koushik Sen, Grigore Rosu, and Gul Agha. Online efficient predictive safety analysis of multithreaded programs. In *TACAS*, pages 123–138, 2004.

[SVAR04] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04), Edinburgh, UK*, pages 418–427. IEEE, May 2004.

[Thi94] P. S. Thiagarajan. A trace based extension of linear time temporal logic. In Samson Abramsky, editor, *Proceedings of the Ninth Annual IEEE Symp. on Logic in Computer Science, LICS 1994*, pages 438–447. IEEE Computer Society Press, July 1994.

[TW02] P. S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for mazurkiewicz traces. *Inf. Comput.*, 179(2):230–249, 2002.

[Val93] Antti Valmari. On-the-fly verification with stubborn sets. In *CAV*, pages 397–408, 1993.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.