Partial Order Techniques for Distributed Discrete Event Systems: why you can't avoid using them

Eric Fabre, Albert Benveniste

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Partial Order Techniques for Distributed Discrete Event Systems: why you can't avoid using them

Eric Fabre — Albert Benveniste

N° 5916 — version 3

version initiale May 2006 — version révisée February 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5916--FR+ENG

Partial Order Techniques for Distributed Discrete Event Systems: why you can't avoid using them*

Eric Fabre , Albert Benveniste †

Thème COM — Systèmes communicants
Projet DistribCom

Rapport de recherche n° 5916 — version 3‡ — version initiale May 2006 — version révisée February 2007 — 53 pages

Abstract: Monitoring or diagnosis of large scale distributed Discrete Event Systems with asynchronous communication is a demanding task. Ensuring that the methods developed for Discrete Event Systems properly scale up to such systems is a challenge. In this paper we explain why the use of partial orders cannot be avoided in order to achieve this objective. To support this claim, we try to push classical techniques (parallel composition of automata and languages) to their limits and we eventually discover that partial order models pop up at some point.

We focus on on-line techniques, where a key difficulty is the choice of proper data structures to represent the set of all runs of a distributed system, in a modular way. We discuss the use of previously known structures such as execution trees and unfoldings. We propose a novel and more compact data structure called "trellis". Then, we show how all the above data structures can be used in performing distributed monitoring and diagnosis.

The techniques reported here were used in an industrial context for fault management and alarm correlation in telecommunications networks.

This paper is an extended and improved version of the plenary address that was given by the second author at WODES'2006.

Key-words: Discrete Event Systems, distributed systems, diagnosis, partial orders, unfoldings, fault management

* This report has been written as a support to the plenary address given by the second author at WODES 2006. This work has been supported in part by joint RNRT contracts Magda and Magda2, with France Telecom R&D and Alcatel, funded by french Ministère de la Recherche, and by direct contracts with Alcatel. This paper reports on experience and joint work with Stefan Haar and Claude Jard, from IRISA. It is based on tight cooperation and interaction with Chistophe Dousson from France Telecom R&D and Armen Aghasaryan from Alcatel.

† IRISA-INRIA, Campus de Beaulieu, 35042 Rennes; surname.name@inria.fr

‡ This is a revised version of the former report. Revision is significant and thus this version should replace the original one.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Techniques d'ordres partiels pour les systèmes à événements discrets répartis: pourquoi ne peut-on s'y soustraire

Résumé : Ce document explique pourquoi l'on ne peut échapper au recours aux modèles d'ordres partiels pour l'algorithmique des systèmes à événements discrets répartis. Dans le présent document nous nous en tenons à la surveillance et ne traitons pas du contrôle. Les techniques présentées ont été utilisées dans un contexte industriel, pour la gestion répartie d'alarmes dans les réseaux de télécommunications.

Mots-clés : systèmes à événements discrets, systèmes répartis, diagnostic, ordres partiels, dépliages, gestion d'alarmes

Contents

1 Introduction

Since the pioneering work by Ramadge and Wonham, the Discrete Event Systems (DES) community has developed a rich body of frameworks, techniques, and algorithms for the supervision of DES. While most authors have considered supervision of a monolithic automaton or language, decentralized frameworks have been more recently considered [5], [7]–[11] and [23, 24], [39]–[41], and [50].

While different architectures have been studied by these authors, the typical situation is the following: The system considered is observed by a finite set of *agents*, indexed by some finite index set I. Agent i can observe events labeled by some subalphabet $L_i \subset L$ of the message alphabet. Local decisions performed by the local agents are then forwarded to some central supervisor, which takes the final decision regarding observation; decisions mechanisms available to the supervisor are simple policies to combine the decisions forwarded by the local agents, *e.g.,* conjunction, disjunction, etc [50, 27]. Of course, there is no reason why such decentralized setting should be equivalent to the centralized one. Therefore, various notions of decentralized observability, controllability, and diagnosability have been proposed for each particular architecture, see *e.g.,* [50]. Deciding upon such properties can then become infeasible [49].

Whereas these are important results, they fail to address the issue of large systems, where global model, global state, and sometimes even global time, should be avoided.

1.1 The problem considered

In this paper, we consider a distributed system $\mathcal{A}$ with subsystems $\mathcal{A}_i, i \in I$ and a set of sensing systems $\mathcal{O}_i, i \in I$ attached to each subsystem. The goal is to perform the monitoring of $\mathcal{A}$ under the following constraints:

- a supervisor $\mathcal{D}_i$ is attached to each subsystem;
- supervisor $\mathcal{D}_i$ does not know the global system model $\mathcal{A}$; it only knows a local view of $\mathcal{A}$, consisting of $\mathcal{A}_i$ plus some interface information relating $\mathcal{A}_i$ to its neighbors;
- supervisor $\mathcal{D}_i$ accesses observations made by $\mathcal{O}_i$;
- the different supervisors act as peers; they can exchange messages with their neighboring supervisors; they concur at performing system monitoring;
- no global clock is available, and the communication infrastructure is asynchronous.

The next issue is to define what we mean by *monitoring*. Usually, the DES and IA communities focus on the problem of diagnosis. Diagnosis consists in detecting and isolating certain *failures* the considered system may be subject to. A failure may be specified as a subset of states, or as the fact of having seen certain events in the history of the system. More generally, a failure can be specified by using appropriate logic formulas that characterize a given set of behaviours, see [26]. The important point in this context is that this set of

failures is typically given in advance. Accordingly, key issues are the algorithm for failure detection and isolation (or diagnosis), as well as diagnosability.

While this is the most commonly addressed problem, it may not be the most relevant one in practice. In appendix B, we describe our experience in terms of industrial collaboration. This reveals that the primary problem was not that of tracking "failures" in the system. The main problem was that of "sorting out" what happened in the system, by using recorded logs, off-line or on-line. When large distributed systems are designed, they typically come up with a distributed pre-defined sensing equipment, be it hardware or software. This sensing equipment typically produces a huge number of low level events. Each event is caused by a certain combination of things that happened here or there to the system. Thus events are very frequently "correlated", meaning that they carry redundant information. Sorting out this mass of information is what the operator expects. Interpretation is then a derived service that may either be left to the human, or be partly or fully automated.

Sorting out information from distributed logs can be performed in many ways. In this tutorial we consider a model based approach and the problem we solve is the following, we call it *distributed monitoring* in the sequel:

> Distributed monitoring: *given logs independently recorded by a distributed set of sensors, what are the possible hidden state histories that are compatible with these logs?* (Call these histories the *solutions* of the monitoring problem in the sequel.)

By "independently", we mean here that the logs are recorded asynchronously, with no central coordination. The above mentioned problem of correlating events or alarms is easily solved once monitoring in the above sense has been performed. Also, failure diagnosis can be seen as a second (although not fully trivial) step following monitoring. Distributed monitoring is in fact the very basis of most distributed tasks related to system observation.

Now, distributed monitoring as defined above is not quite what is needed for very large distributed systems. Such systems are generally decomposed into different *domains* and each domain is managed by its own supervisor. The different supervisors act as peers and concur at managing the entire system in a distributed, unsupervised way. Each supervisor is thus only interested in what is happening within its own domain, it is not concerned with the other domains. Thus, in this case, distributed monitoring should be replaced by *modular* distributed monitoring:

> Modular distributed monitoring: *given logs independently recorded by each supervisor through its local sensors, compute, for each supervision domain, a local view of (global) monitoring.*

Finally, in contrast with most DES studies, we shall not pre-compute the set of all candidate solutions as it is, *e.g.,* , performed in the diagnoser approach [45] to DES diagnosis. We are indeed interested in the set of histories compatible with a given set of logs. This cannot be pre-computed prior to collecting these logs.

1.2 Objectives of this tutorial

While centralized and decentralized diagnosis under synchronous communication are handled in a nice algebraic framework [50, 27], the situation is much less satisfactory when distributed diagnosis under asynchronous communications is considered. We believe that this is mostly due to the lack of a proper algebraic setting to deal with both distribution and asynchrony. Relying on our previous experience in failure diagnosis for telecommunication networks, we have converged to a quite general algebraic framework for these problems, that turns out to share some features with other contributions to the topic [46, 47]. In particular, we have identified three essential features of such a framework:

1. *Factorization issues.* The notion of product is central to express that a large system is obtained by assembling components. For example a distributed system can be expressed as the parallel composition of automata. The parallel composition is also useful to represent the operation of constraining a component to produce the collected observations. Therefore, the parallel composition of automata plays a central role in diagnosis. It is also essential that the solutions to the diagnosis problem be themselves expressible as a product of local solutions: this is the key to modular computations.

2. *Projection issues.* Projecting on a given component the global solutions to the diagnosis problem gives the so-called local view of the diagnosis. A key feature of decentralized approaches is to compute these local views directly, without computing the possibly huge global solutions. This is done by suitable combinations of products and projections, provided these two operations jointly satisfy a few axioms. Therefore projections must be designed with care.

3. *Efficient data structures.* Distributed computations must handle sets of trajectories. Most DES approaches represent them as languages. But these data structures do not scale up, even under their factorized form (based on the shuffle product). So a crucial issue is to represent sets of runs in the most efficient way, while preserving factorization properties, and ensuring the existence of adequate projections.

If these three issues are not properly handled, distributed diagnosis algorithms become rapidly intractable and have little chance to scale up. In this tutorial we study these three aspects: Section 3 is devoted to issue 1, whereas Section 4 focuses on issues 2 and 3.

Interestingly enough, although basing all developments on the usual sequential semantics of DES, we will show that partial order models will inevitably pop up, under the form of a distributed notion of time. This strongly suggests that the adequate manner to handle distributed systems is to take explicitly into account the concurrency of events. Section 5 is devoted to this important issue.

2 Formal problem setting: monitoring in terms of runs

In this section we formalize distributed monitoring. Our basic setting is classical and uses automata and their languages. Our system for monitoring is modelled as an automaton

$$\mathcal{A} = (S, L, \rightarrow, S_0),$$

where S is the set of states, L is the set of labels, $\rightarrow \subseteq S \times L \times S$ is the transition relation, and $S_0 \subseteq S$ is the set of initial states. Write $s \xrightarrow{\ell} s'$ to mean that $(s, \ell, s') \in \rightarrow$. Call *run* a finite or infinite sequence of successive transitions:

$$\sigma \quad : \quad s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} s_2 \ldots, \text{ where } s_0 \in S_0, \tag{1}$$

and denote by $\Sigma_{\mathcal{A}}$ the set of all runs of $\mathcal{A}$. Recall the *weakly synchronous product* of automata, also called "parallel composition" in DES litterature:

$$\mathcal{A}_1 \times \mathcal{A}_2 \quad = \quad (S_1 \times S_2, L_1 \cup L_2, \rightarrow, S_{0,1} \times S_{0,2}) \tag{2}$$

where $(s_1, s_2) \xrightarrow{\ell} (s_1', s_2')$ iff the automata progress either locally (cases (i) and (iii)) or jointly (case (ii)):

$$\begin{array}{lccccc} \text{(i)} & \ell \in L_1 \setminus L_2 & \wedge & s_1 \xrightarrow{\ell}_1 s_1' & \wedge & s_2' = s_2 \\ \text{(ii)} & \ell \in L_2 \cap L_1 & \wedge & s_1 \xrightarrow{\ell}_1 s_1' & \wedge & s_2 \xrightarrow{\ell}_2 s_2' \\ \text{(iii)} & \ell \in L_2 \setminus L_1 & \wedge & s_1' = s_1 & \wedge & s_2 \xrightarrow{\ell}_2 s_2' \end{array}$$

Product (2) is commutative and associative.

For $\mathcal{A} = (S, L, \rightarrow, S_0)$ an automaton, its language $\mathcal{L}_{\mathcal{A}}$ is the set of words over alphabet L that its set of runs $\Sigma_{\mathcal{A}}$ generates; note that $\mathcal{L}_{\mathcal{A}}$ is prefix closed.

If $\mathcal{L}$ is a language over alphabet L, let $\mathbf{proj}_{L,L'}(\mathcal{L})$ denote the *projection* of $\mathcal{L}$ over L', obtained by erasing, in any word of $\mathcal{L}$, all labels not belonging to L' (we do not require $L' \subseteq L$). For $\mathcal{L}'$ a language over L', the *inverse projection* $\mathbf{proj}^{-1}_{L,L'}(\mathcal{L}')$ is the set of words w over L such that $\mathbf{proj}_{L,L'}(w) \in \mathcal{L}'$. When no confusion can occur, we shall feel free to write

$$\begin{array}{c} \mathbf{proj}_{L'}(\mathcal{L}) \text{ for short instead of } \mathbf{proj}_{L,L'}(\mathcal{L}), \\ \text{and} \quad \mathbf{proj}^{-1}_{L}(\mathcal{L}') \text{ for short instead of } \mathbf{proj}^{-1}_{L,L'}(\mathcal{L}'). \end{array}$$

The *shuffle product* of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ defined over alphabets L_1 and L_2 is the following language defined over $L = L_1 \cup L_2$:

$$\mathcal{L}_1 \times_L \mathcal{L}_2 \quad = \quad \mathbf{proj}^{-1}_{L}(\mathcal{L}_1) \cap \mathbf{proj}^{-1}_{L}(\mathcal{L}_2). \tag{3}$$

It satisfies:

$$\mathcal{L}_{\mathcal{A}_1 \times \mathcal{A}_2} \quad = \quad \mathcal{L}_{\mathcal{A}_1} \times_L \mathcal{L}_{\mathcal{A}_2}. \tag{4}$$

Let $\mathcal{A} = (S, L, \to, S_0)$ be an automaton. Partition L as $L = L_o \cup L_u$, where L_o and L_u are the subsets of *observed* and *unobserved* labels, respectively. Let

$$\mathcal{L}_{\mathcal{A},o} \quad =_{\text{def}} \quad \mathbf{proj}_{L_o}(\mathcal{L}_{\mathcal{A}})$$

be the *observed language* of $\mathcal{A}$. Let $\overline{\mathbf{proj}}_{A,L_o} : \Sigma_{\mathcal{A}} \mapsto \mathcal{L}_{\mathcal{A},o}$ be the map associating, to each run $\sigma \in \Sigma_{\mathcal{A}}$, the observation $\omega \in \mathcal{L}_{\mathcal{A},o}$ it generates. Call *monitor* of $\mathcal{A}$ the reverse map:

$$\mathcal{L}_{\mathcal{A},o} \ni \omega \quad \longmapsto \quad \overline{\mathbf{proj}}^{-1}_{A,L_o}(\omega) \subseteq \Sigma_{\mathcal{A}}. \tag{5}$$

In words, the monitor of $\mathcal{A}$ is any algorithm that computes, for every observation $\omega \in \mathcal{L}_{\mathcal{A},o}$, the set of runs compatible with ω, we call them also the set of runs *explaining* ω. Note that this set is not empty, since we assume that ω itself was generated by some actual run of $\mathcal{A}$. Map (5) extends to observations that are themselves sets of observations: $2^{\mathcal{L}_{\mathcal{A},o}} \mapsto 2^{\Sigma_{\mathcal{A}}}$. Sets of observations will be generically denoted by Ω. Hence our extended definition for the monitor:

Definition 1 *The* monitor *of $\mathcal{A}$ is the map:*

$$\mathcal{L}_{\mathcal{A},o} \supseteq \Omega \quad \longmapsto \quad \overline{\mathbf{proj}}^{-1}_{A,L_o}(\Omega) \subseteq \Sigma_{\mathcal{A}}. \tag{6}$$

Returning to our requirements of Section 1.1, we assume that the automaton for monitoring, as well as its corresponding observations, decompose as

$$\mathcal{A} \quad = \quad \times_{i\in I}\, \mathcal{A}_i \tag{7}$$

$$\Omega \quad = \quad \times^{L}_{i\in I}\, \omega_i \tag{8}$$

where

$$\begin{aligned} \mathcal{A}_i \quad &= \quad (S_i, L_i, \to_i, S_{i,0}),\ L_i = L_{o,i} \cup L_{u,i}, \\ L \quad &= \quad L_o \cup L_u,\ \text{with } L_o = \bigcup_{i\in I} L_{o,i}, \\ \text{and } \omega_i \quad &\in \quad \mathcal{L}_{\mathcal{A}_i,o} \text{ is a singleton (local) observation for } \mathcal{A}_i. \end{aligned}$$

In formula (8), the shuffle product makes the resulting global observation a language, not a singleton. In particular, when the observed alphabets for the different sites i are pairwise disjoint, Ω is the set of all possible interleavings of the local observations — this reflects the independence and asynchrony of the distributed sensors. To summarize, our problem is the following:

Problem 1 (distributed monitoring) *Find a distributed monitoring algorithm for system (7) and (8), where supervisors $\mathcal{D}_i$, respectively attached to each site $i \in I$, concur at computing the monitor (6), by exchanging messages, asynchronously and in an unsupervised way.*

3 Distributed monitoring in terms of languages

3.1 Weakened formulation: monitoring in terms of languages

In order to present the essential techniques of distributed monitoring algorithms, we shall first consider a weakened formulation of Problem 1. Instead of asking for the reconstruction of all runs explaining an observation, we shall only ask for the reconstruction of the sublanguage of $\mathcal{L}_{\mathcal{A}}$ that can explain the observations (this is a weaker problem unless $\mathcal{A}$ is a deterministic automaton). This problem was first considered by Su and Wonham and extensively studied in [46, 47, 48]. The approach we present here aims at preparing for other data structures to encode solutions.

Definition 2 (monitor) *The* language monitor *of $\mathcal{A}$ is the map:*

$$\mathcal{L}_{\mathcal{A},o} \supseteq \Omega \quad \longmapsto \quad \mathbf{proj}_L^{-1}(\Omega) \subseteq \mathcal{L}_{\mathcal{A}}. \tag{9}$$

Note that we do not assume that Ω is prefix closed. So, neither is the solution $\mathbf{proj}_L^{-1}(\Omega)$ of language monitoring. This expresses the fact that the solutions must explain the entire observation, not just a prefix of it.

What makes Definition 2 simpler to handle than Definition 1 is the fact that the language monitor maps languages to languages, instead of languages to sets of runs (traversed states are omitted). This will allow for a more algebraic reformulation of the language monitoring problem. In particular, for Ω a set of observations, we have

$$\mathbf{proj}_L^{-1}(\Omega) \quad = \quad \mathcal{L}_{\mathcal{A}} \times_L \Omega$$

Now, considering again our distributed setting (7) and (8), language monitoring consists in computing

$$\mathcal{L}_{(\times_{i\in I}\mathcal{A}_i)} \times_L \left(\bigtimes_{i\in I}^L \omega_i\right)$$

By (4), we have

$$\mathcal{L}_{(\times_{i\in I}\mathcal{A}_i)} \times_L \left(\bigtimes_{i\in I}^L \omega_i\right) \quad = \quad \bigtimes_{i\in I}^L \left(\mathcal{L}_{\mathcal{A}_i} \times_L \omega_i\right)$$

and, therefore, Problem 1 is replaced by the following weaker problem:

Problem 2 *Compute the map*

$$(\omega_i)_{i\in I} \quad \longmapsto \quad \mathcal{M} =_{\text{def}} \bigtimes_{i\in I}^L \left(\mathcal{L}_{\mathcal{A}_i} \times_L \omega_i\right) \tag{10}$$

in a distributed, asynchronous, and unsupervised way.

In this section we address Problem 2 for the following two cases: off-line monitoring with finite observations, and on-line monitoring for non terminating observations. At this point, note that Problem 2 still involves computing the global solution to monitoring, not the local views for it. We come to the latter in the following section.

3.2 A factorization result; application to modular monitoring

In this section we collect some results on the relations between automata, their languages, and compositions and projections thereof. Even though these results may seem really trivial and routine, we insist listing them. The reason is that these will also constitute the key steps in getting distributed monitoring algorithms when using more efficient data structures. In the latter case, however, those trivial facts will not be trivial any more.

Recall the following standard operations on languages that we shall consistently use in the sequel: 1/ the *intersection* $\mathcal{L} \cap \mathcal{L}'$, for $\mathcal{L}$ and $\mathcal{L}'$ two languages over the same alphabet L; 2/ the *projection* $\mathbf{proj}_{L'}(\mathcal{L})$, for $\mathcal{L}$ a language over alphabet L and L' another alphabet; and, 3/ the *product* $\mathcal{L} \times_L \mathcal{L}'$ (the shuffle product of languages). The following (trivial looking) result will be instrumental in getting our distributed asynchronous monitoring algorithms:

Theorem 1 (factorization) *We are given a product $\mathcal{A} = \times_{i\in I} \mathcal{A}_i$ of automata. For each $i \in I$, let $\mathcal{L}_i$ be a sublanguage of $\mathcal{L}_{\mathcal{A}_i}$, and let $\mathcal{L} =_{\rm def} \times^L_{i\in I} \mathcal{L}_i$ be their product. Then:*

1. *We have $\mathcal{L}_{\mathcal{A}} = \times^L_{i\in I} \mathcal{L}_{\mathcal{A}_i}$.*
2. *$\mathcal{L}$ is a sublanguage of $\mathcal{L}_{\mathcal{A}}$ and, for all $i \in I$, $\mathbf{proj}_{L_i}(\mathcal{L})$ is a (generally strict) sublanguage of $\mathcal{L}_i$.*
3. *We have $\mathcal{L} = \times^L_{i\in I} \mathbf{proj}_{L_i}(\mathcal{L})$. Furthermore, $\mathcal{L}^\star_i =_{\rm def} \mathbf{proj}_{L_i}(\mathcal{L})$ yields the* minimal *decomposition of $\mathcal{L}$ in that, for any decomposition $\mathcal{L} = \times^L_{i\in I} \mathcal{L}'_i$, where $\mathcal{L}'_i$ is a sublanguage of $\mathcal{L}_{\mathcal{A}_i}$, then $\mathcal{L}^\star_i$ is a sublanguage of $\mathcal{L}'_i$.*

Using Theorem 1, Problem 2 is subsumed by the following one:

Problem 3 *Let $\mathcal{L}_i, i \in I$ be a finite set of languages defined over alphabets $L_i = L_{o,i} \uplus L_{u,i}$. Compute the map*

$$(\omega_i)_{i\in I} \quad \longmapsto \quad \mathcal{M} =_{\rm def} \times^L_{i\in I} \mathbf{proj}_{L_i}(\mathcal{L}_{\mathcal{A}} \times_L \Omega), \tag{11}$$

where $\Omega = \times^L_{i\in I} \omega_i$ and ω_i ranges over $\mathbf{proj}_{L_{o,i}}(\mathcal{L}_i)$.

Now, as discussed in Section 1.1, we are not really interested in computing global monitoring, as performed by formula (11). We are rather only interested in computing consistent local views of global monitoring, *i.e.,* for each $i \in I$, the local projection $\mathbf{proj}_{L_i}(\mathcal{L}_{\mathcal{A}} \times_L \Omega)$. This was referred to as *modular* distributed monitoring in Section 1.1. Consequently, we can further subsume Problem 3 by the following problem, called *modular* language monitoring:

Problem 4 (modular language monitoring) *Let $\mathcal{L}_i, i \in I$ be a finite set of languages defined over alphabets $L_i = L_{o,i} \uplus L_{u,i}$. Compute the map*

$$(\omega_i)_{i\in I} \quad \longmapsto \quad \mathcal{M}_{\rm mod} =_{\rm def} \left(\mathbf{proj}_{L_i}(\mathcal{L}_{\mathcal{A}} \times_L \Omega)\right)_{i\in I}, \tag{12}$$

where $\Omega = \times^L_{i\in I} \omega_i$ and ω_i ranges over $\mathbf{proj}_{L_{o,i}}(\mathcal{L}_i)$.

We shall focus on solving Problem 4 and its variants in the sequel.

3.3 Four basic objectives

The following basic objectives must be addressed, we shall do this in the sequel:

Objective 1 *Address asynchronous distributed systems with unsupervised supervising peers. This requires computing computing $\mathcal{M}_{\rm mod}$ without computing $\mathcal{M}$, by attaching a supervising peer to each site.*

Objective 2 *Compute $\mathcal{M}_{\rm mod}$ on-line and on the fly.*

Objective 3 *Avoid state explosion due to the concurrency between and possibly within the different components.*

Objective 4 *Address changes in the systems dynamics.*

3.4 Distributed modular monitoring — Objective 1

Getting distributed algorithms for modular monitoring relies on the following fundamental result, which shows how to compute modular monitoring locally, for simple basic cases:

Theorem 2 *Let $(\mathcal{L}_i)_{i=1,2,3}$ be three languages such that*

$$(L_1 \cap L_3) \subseteq L_2 \qquad (L_2 \text{ separates } L_1 \text{ from } L_3)$$

Write $\mathbf{proj}_i(.)$ for short instead of $\mathbf{proj}_{L_i}(.)$. Then, the following formulas hold:

$$\mathbf{proj}_2(\mathcal{L}_1 \times_L \mathcal{L}_2 \times_L \mathcal{L}_3) = \underbrace{\underbrace{\mathbf{proj}_2(\mathcal{L}_1)}_{\text{local to 1}} \cap \mathcal{L}_2 \cap \underbrace{\mathbf{proj}_2(\mathcal{L}_3)}_{\text{local to 3}}}_{\text{local to 2}} \tag{13}$$

$$\mathbf{proj}_1(\mathcal{L}_1 \times_L \mathcal{L}_2 \times_L \mathcal{L}_3) = \underbrace{\mathcal{L}_1 \cap \underbrace{\mathbf{proj}_1(\mathcal{L}_2 \cap \underbrace{\mathbf{proj}_2(\mathcal{L}_3)}_{\text{local to 3}})}_{\text{local to 2}}}_{\text{local to 1}} \tag{14}$$

Note that the intersections in formulas (13) and (14) are in fact products, since the involved alphabets are identical. A direct induction reasoning regarding the cardinal of index set J allows us to extend formula (13) to an arbitrary number of languages as follows:

Corollary 1 *Let $(\mathcal{L}_j)_{j\in J}$ be any family of languages such that there exists $j_o \in J$ such that:*

$$\forall (j,j') \in J \times J \ : \ j_o \neq j \neq j' \neq j_o \quad \Rightarrow \quad L_j \cap L_{j'} \subseteq L_{j_o} \tag{15}$$

Then,

$$\mathbf{proj}_{j_o}(\bigtimes^L_{j\in J} \mathcal{L}_j) = \mathcal{L}_{j_o} \cap \big(\bigcap_{j\in J, j\neq j_o} \mathbf{proj}_{j_o}(\mathcal{L}_j)\big) \tag{16}$$

Regarding Theorem 2, a direct proof is easily obtained if we remember that the words of $\mathcal{L} \times_L \mathcal{L}'$ are obtained by synchronizing the words of $\mathcal{L}$ and the words of $\mathcal{L}'$. However, such a direct proof would not easily generalize to the stronger monitoring problem of Section 2, and would not generalize either to more efficient data structures. To prepare

for such a generalization, we shall base the proof of Theorem 2 on the following lemma.

Lemma 1 *For L' any alphabet, $\mathcal{L}$ any language over arbitrary alphabet L, and $\mathcal{L}_i$ any language over arbitrary alphabet L_i, the following properties hold:*

$$\begin{aligned} \mathbf{proj}_{L_1} \circ \mathbf{proj}_{L_2}(\mathcal{L}) &= \mathbf{proj}_{L_1 \cap L_2}(\mathcal{L}) && (17)\\ \mathbf{proj}_{L}(\mathcal{L}) &= \mathcal{L} && (18)\\ L_3 \supseteq (L_1 \cap L_2) \;\Rightarrow\; \mathbf{proj}_{L_3}(\mathcal{L}_1 \times_L \mathcal{L}_2) &= \mathbf{proj}_{L_3}(\mathcal{L}_1) \times_L \mathbf{proj}_{L_3}(\mathcal{L}_2) \\ &= \mathbf{proj}_{L_3}(\mathcal{L}_1) \cap \mathbf{proj}_{L_3}(\mathcal{L}_2) && (19) \end{aligned}$$

In (17), symbol $\circ$ stands for the composition of maps; thus we have: $\mathbf{proj}_{L_1} \circ \mathbf{proj}_{L_2}(\mathcal{L}) = \mathbf{proj}_{L_1}\big(\mathbf{proj}_{L_2}(\mathcal{L})\big)$. Also, (19) generalizes to more than two languages, in a similar way as we did in Corollary 1.

Proof: We only need to prove (19) as the other properties are trivial. Set $L = L_1 \cup L_2$ and pick a pair $(w_1, w_2) \in \mathcal{L}_1 \times \mathcal{L}_2$. Then, there exists $w \in \mathcal{L}$ such that $w_i = \mathbf{proj}_{L_i}(w)$ for $i = 1, 2$, if and only if $\mathbf{proj}_{L_1 \cap L_2}(w_1) = \mathbf{proj}_{L_1 \cap L_2}(w_2)$. But, since $L_3 \supseteq L_1 \cap L_2$, this condition rewrites:

$$\mathbf{proj}_{L_1 \cap L_2}\big(\mathbf{proj}_{L_3}(w_1)\big) \;=\; \mathbf{proj}_{L_1 \cap L_2}\big(\mathbf{proj}_{L_3}(w_2)\big) \qquad (20)$$

For $i = 1, 2$, set $w'_i = \mathbf{proj}_{L_3}(w_i)$. Then, (20) holds if and only if the pair (w'_1, w'_2) yields a word $w' \in \mathbf{proj}_{L_3}(\mathcal{L}_1) \times_L \mathbf{proj}_{L_3}(\mathcal{L}_2)$ such that $w'_i = \mathbf{proj}_{L_i}(w')$. This shows (19). ◇

Proof of Theorem 2: We first prove (13). Since $L_2 \supseteq (L_1 \cap L_3)$, we have

$$\begin{aligned} \mathbf{proj}_2(\mathcal{L}_1 \times_L \mathcal{L}_2 \times_L \mathcal{L}_3) &\overset{\text{by (19)}}{=} \mathbf{proj}_2(\mathcal{L}_1) \cap \mathbf{proj}_2(\mathcal{L}_2 \times_L \mathcal{L}_3) \\ &\overset{\text{by (19,18)}}{=} \mathbf{proj}_2(\mathcal{L}_1) \cap \mathcal{L}_2 \cap \mathbf{proj}_2(\mathcal{L}_3) \end{aligned}$$

To prove (14), note that $L_2 \supseteq (L_1 \cap L_3)$ implies $L_1 \cup L_2 \supseteq L_1 \cap (L_2 \cup L_3)$, hence

$$\begin{aligned} \mathbf{proj}_1(\mathcal{L}_1 \times_L \mathcal{L}_2 \times_L \mathcal{L}_3) &\overset{\text{by (19)}}{=} \mathbf{proj}_1\big(\mathbf{proj}_{L_1 \cup L_2}(\mathcal{L}_1) \times_L \mathbf{proj}_{L_1 \cup L_2}(\mathcal{L}_2 \times_L \mathcal{L}_3)\big) \\ &\overset{\text{by (19)}}{=} \mathbf{proj}_1\big(\mathbf{proj}_{L_1 \cup L_2}(\mathcal{L}_1) \times_L \mathbf{proj}_{L_1 \cup L_2}(\mathcal{L}_2) \times_L \mathbf{proj}_{L_1 \cup L_2}(\mathcal{L}_3)\big) \\ &\overset{\text{by (17,18)}}{=} \mathbf{proj}_1\big(\mathcal{L}_1 \times_L \mathcal{L}_2 \times_L \mathbf{proj}_2(\mathcal{L}_3)\big) \\ &\overset{\text{by (19)}}{=} \mathcal{L}_1 \cap \mathbf{proj}_1\big(\mathcal{L}_2 \cap \mathbf{proj}_2(\mathcal{L}_3)\big) \end{aligned}$$

which proves the theorem. The key point is that only Lemma 1 was used in its proof. ◇

Building blocks for the distributed algorithms. Define the following operators, attached to the pair of sites (i,j) and site i, respectively:

$$\mathbf{Msg}_{i\to j}(\mathcal{V}_i) \quad =_{\text{def}} \quad \text{compute } \mathbf{proj}_j(\mathcal{V}_i) \text{ at site } i \text{ and send the result to site } j \tag{21}$$

$$\mathbf{Fuse}[\mathcal{V}_i, \mathcal{V}_i'] \quad =_{\text{def}} \quad \text{compute } \mathcal{V}_i \cap \mathcal{V}_i' \text{ at site } i \tag{22}$$

As a result of performing $\mathbf{Msg}_{i\to j}(\mathcal{V}_i)$, the projection $\mathbf{proj}_j(\mathcal{V}_i)$ can be used by site j for subsequent operations, see Algorithm 1 below. Notice that the **Fuse** operator generalizes to any number of messages. Using these operators, rules (13) and (14) respectively rewrite as

$$\mathbf{proj}_2(\mathcal{V}_1 \times_L \mathcal{V}_2 \times_L \mathcal{V}_3) \quad = \quad \mathbf{Fuse}[\mathbf{Msg}_{1\to 2}(\mathcal{V}_1), \mathcal{V}_2, \mathbf{Msg}_{3\to 2}(\mathcal{V}_3)] \tag{23}$$

$$\mathbf{proj}_1(\mathcal{V}_1 \times_L \mathcal{V}_2 \times_L \mathcal{V}_3) \quad = \quad \mathbf{Fuse}[\mathcal{V}_1, \mathbf{Msg}_{2\to 1}(\mathbf{Fuse}[\mathcal{V}_2, \mathbf{Msg}_{3\to 2}(\mathcal{V}_3)])] \tag{24}$$

The following obvious lemma will be instrumental in developing our distributed, unsupervised, and asynchronous algorithms:

Lemma 2 *The two maps*

$$\begin{aligned} \mathcal{V}_i &\mapsto \mathbf{Msg}_{i\to j}(\mathcal{V}_i) \\ (\mathcal{V}_i, \mathcal{V}_i') &\mapsto \mathbf{Fuse}[\mathcal{V}_i, \mathcal{V}_i'] \end{aligned}$$

where $j \in I$ is arbitrary, are increasing w.r.t. all their arguments, for the order of language inclusion. Furthermore, $\mathbf{Fuse}[\mathcal{V}_i, \mathcal{V}_i'] \subseteq \mathcal{V}_i$.

Message passing algorithm with chaotic iterations. Let $(\mathcal{L}_i)_{i\in I}$ be a collection of languages. We wish to design a distributed algorithm for

$$\text{computing } \mathbf{proj}_j(\mathcal{L}) \text{ for each } j \in I, \text{ where } \mathcal{L} = \bigtimes_{i\in I}^{L} \mathcal{L}_i.$$

We shall propose a distributed algorithm, in which supervisors act as peers, by exchanging messages asynchronously. Since this algorithm is by message passing, the topology of the communications graph between the peers plays a role. Thus we formalize this now. Define the *interaction graph* $\mathcal{G}_I$ of $(\mathcal{L}_i)_{i\in I}$ as the following non directed graph:

vertices of $\mathcal{G}_I$ are labeled with the indices $i \in I$, and
(i,j) is a branch of $\mathcal{G}_I$ if and only if $i \neq j$ and $L_i \cap L_j \neq \emptyset$.

Say that *node j separates two nodes i,k* on the graph $\mathcal{G}_I$ if every path leading i to k must include node j. In particular, if j separates nodes i,k, then L_k separates L_i from L_j in the sense of Theorem 2. Fig. 1 shows an example where the interaction graph is a tree.

We are now ready to state our first algorithm. In this algorithm, the different supervisors act as peers exchanging messages asynchronously. For this, we assume that buffers are available in the two directions, for each branch of the interaction graph. Each supervisor i writes its successive messages to its neighbour j in its outgoing buffer toward j, and reads its messages from neighbour j from its incoming buffer from j. Reads and writes are asynchronous.

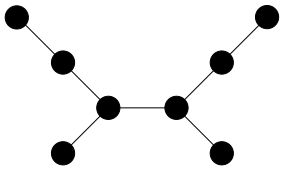

Figure 1: An example of system where the interaction graph $\mathcal{G}_I$ is a tree.

Algorithm 1 (chaotic message passing) *The different supervising peers, respectively attached to each site $i \in I$, maintain and exchange messages $\mathcal{M}_{i\to j}$ with their neighbours j, where $(j,i) \in \mathcal{G}_I$, by performing, independently and in a chaotic way:*

- Initialization: *each peer $i \in I$ initializes its set of messages as follows:*

$$\forall (i,j) \in \mathcal{G}_I \quad : \quad \mathcal{M}_{i\to j} := (L_i \cap L_j)^* \tag{25}$$

- Chaotic iteration step: *each peer $i \in I$ performs:*

  1. *pick $(i,j) \in \mathcal{G}_I$, define $J_{ij} =_{\text{def}} \{k \in I \mid (i,k) \in \mathcal{G}_I, k \neq j\}$;*
  2. *read the current value of $\mathcal{M}_{k\to i}$, for $k \in J_{ij}$;*
  3. *update message to peer j by performing the following two operations, in sequence:*

$$\mathcal{M}_{i\to j} \quad := \quad \mathbf{Msg}_{i\to j} \circ \mathbf{Fuse}\left[\mathcal{L}_i \times_L \omega_i\,,\, \{\mathcal{M}_{k\to i} \mid k \in J_{ij}\}\right] \tag{26}$$

  4. *read the current value of $\mathcal{M}_{l\to i}$, for $(i,l) \in \mathcal{G}_I, l \neq i$ and update*

$$\mathcal{L}'_i \quad := \quad \mathbf{Fuse}\left[\mathcal{L}_i \times_L \omega_i\,,\, \{\mathcal{M}_{l\to i} \mid (i,l) \in \mathcal{G}_I, l \neq i\}\right] \tag{27}$$

The above steps *1–4* are performed chaotically by each supervisor, acting as a peer. They do not need to be performed in an atomic way, *i.e.,* while performing them, the different peers do not block each other. In fact, we shall later see that update (27) need not be performed for each step of the chaotic algorithm, but only at its termination (see the discussion on termination following the proof of Theorem 3). The resulting chaotic message passing algorithm is thus completely distributed, unsupervised, and asynchronous. Note that Algorithm 1 is non terminating in the sense that no stopping criterion is formulated for it.

Definition 3 *Say that the branch $(i,j) \in \mathcal{G}_I$ is* fairly visited *by Algorithm 1 if it is selected infinitely many times while performing (26). Say that chaotic Algorithm 1 is* fairly executed *if each branch $(i,j) \in \mathcal{G}_I$ is fairly visited.*

Theorem 3 *Assume that the interaction graph $\mathcal{G}_I$ is a tree. Then, chaotic message passing Algorithm 1 converges in the following sense: if the algorithm is fairly executed, then the sequence of successive updates of $\mathcal{L}'_i$ by (27) is decreasing (for the sublanguage order) and converges to the desired solution $\mathbf{proj}_i(\mathcal{L} \times_L \Omega)$.*

Proof: To simplify notations for this proof, we shall rename $\mathcal{L}_i \times_{_L} \omega_i$ as $\mathcal{L}_i$ and $\mathcal{L} \times_{_L} \Omega$ as $\mathcal{L}$; thus we want to prove that the sequence of successive updates of $\mathcal{L}'_i$ converges to $\mathbf{proj}_i(\mathcal{L})$. The key idea for the proof consists in marking the messages $\mathcal{M}_{i\to j}$ with the subset of sites K_{ij} that this message takes into account:

$$\boldsymbol{M}_{i\to j} \quad = \quad (\mathcal{M}_{i\to j}, K_{ij}).$$

To this end, enhance the **Msg** and **Fuse** operations with additional marks $K, K' \subseteq I$ as follows:

$$\begin{array}{rcl} \boldsymbol{Msg}_{i\to j}\,(\mathcal{L}, K) & =_{\text{def}} & \text{send to site } j \text{ the pair } : \big(\mathbf{proj}_j(\mathcal{L}), K\big) \\ \boldsymbol{Fuse}\,[\,(\mathcal{L}, K), (\mathcal{L}', K')\,] & =_{\text{def}} & \big(\mathbf{Fuse}\,[\,\mathcal{L}, \mathcal{L}'\,],\ K \cup K'\big) \end{array}$$

and rewrite Algorithm 1 as follows:

Algorithm 2 (chaotic message passing, with marks) *The different supervising peers, respectively attached to each site $i \in I$, perform, independently and in a chaotic way:*

- Initialization: *each peer $i \in I$ initializes its set of messages as follows:*

$$\forall (i,j) \in \mathcal{G}_I \quad : \quad \boldsymbol{M}_{i\to j} := ((L_i \cap L_j)^*, \emptyset) \tag{28}$$

- Chaotic iteration step: *each peer $i \in I$ performs:*

  1. *pick $(i,j) \in \mathcal{G}_I$ and define $J_{ij} \subseteq \{k \in I \mid (i,k) \in \mathcal{G}_I, k \neq j\}$;*
  2. *read the current value of $\boldsymbol{M}_{k\to i}$, for $k \in J_{ij}$;*
  3. *update message to peer j:*

$$\boldsymbol{M}_{i\to j} \quad := \quad \boldsymbol{Msg}_{i\to j} \circ \boldsymbol{Fuse}\,[\,(\mathcal{L}_i, \{i\}), \{\,\boldsymbol{M}_{k\to i} \mid k \in J_{ij}\}\,] \tag{29}$$

  4. *read the current value of $\boldsymbol{M}_{l\to i}$, for $(i,l) \in \mathcal{G}_I, l \neq i$ and update*

$$(\mathcal{L}'_i, K_i) \quad := \quad \boldsymbol{Fuse}\,[\,(\mathcal{L}_i, \{i\}), \{\,\boldsymbol{M}_{l\to i} \mid (i,l) \in \mathcal{G}_I, l \neq i\}\,] \tag{30}$$

The proof of Theorem 3 is now based on Algorithm 2 with marks and proceeds by induction. Assume that, at some point, the following holds, for each branch $(i,j) \in \mathcal{G}_I$:

$$\boldsymbol{M}_{i\to j} = (\mathcal{M}_{i\to j}, K_{ij}) \quad \Rightarrow \quad \mathcal{M}_{i\to j} \ = \ \mathbf{proj}_{ij}\big(\times^L_{k\in K_{ij}}\, \mathcal{L}_k\big). \tag{31}$$

where $\mathbf{proj}_{ij} =_{\text{def}} \mathbf{proj}_{L_i \cap L_j}$. Fix a branch (i,j) and apply iteration step (29) with this branch. This yields the update

$$\boldsymbol{M}'_{i\to j} \quad = \quad (\mathcal{M}'_{i\to j}, K'_{ij}),$$

where

$$\begin{aligned}
\mathcal{M}'_{i\to j} &= \mathbf{proj}_{ij}(\mathcal{L}_i \cap \bigcap_{k\in J_{ij}} \mathcal{M}_{k\to i}) \\
&\overset{\text{by (31)}}{=} \mathbf{proj}_{ij}(\mathcal{L}_i \cap \bigcap_{k\in J_{ij}} \mathbf{proj}_{ki}(\times^L_{l\in K_{ki}} \mathcal{L}_l)) \\
&\overset{\text{by (17)}}{=} \mathbf{proj}_{ij}(\mathbf{proj}_i(\mathcal{L}_i \cap \bigcap_{k\in J_{ij}} \mathbf{proj}_i(\times^L_{l\in K_{ki}} \mathcal{L}_l))) \\
&\overset{\text{by (16)}}{=} \mathbf{proj}_{ij}(\mathbf{proj}_i(\mathcal{L}_i \cap \times^L_{l\in\bigcup_{k\in J_{ij}} K_{ki}} \mathcal{L}_l)) \\
&\overset{\text{by (17)}}{=} \mathbf{proj}_{ij}(\mathcal{L}_i \cap \times^L_{l\in\bigcup_{k\in J_{ij}} K_{ki}} \mathcal{L}_l) \qquad (32)
\end{aligned}$$

and

$$K'_{ij} = \{i\} \cup \bigcup_{k\in J_{ij}} K_{ki} \qquad (33)$$

Note that, in applying (16), we have used the fact that node i pairwise separates the subsets K_{ki}, for $k \in J_{ij}$, which in turn holds because $\mathcal{G}_I$ is a tree. Comparing (32) and (33) shows that (31) holds for $\boldsymbol{M}'_{i\to j}$. This proves the induction step. On the other hand, (31) holds after the first application of (30), with $K_{ij} = \emptyset$. Thus, property (31) is an invariant of Algorithm 2.

The proof of the theorem follows by noticing that, when updated as in (33),

$$\{i\} \cup \bigcup_{(i,j)\in\mathcal{G}_I} K_{ji}$$

converges to the entire set I for each $i \in I$, if and only if the algorithm is fairly executed. ⋄

Algorithm 2 is illustrated in Fig. 2. We insist that the marking of the messages with the K_i's is only for the purpose of the proof. It need not be implemented in practice, since updating the $\mathcal{L}'_i$'s makes no explicit use of the K_i's.

Termination of the algorithm. In fact, updating (30) need not to be performed at each step, since $\mathcal{L}'_i$ is never used to update the messages that circulate. Strictly speaking, it is enough to perform (30) when the algorithm has terminated, *i.e.,* when the messages have converged to a steady value.

The proof of Theorem 3 shows in passing that convergence occurs in finitely many steps. A natural termination criterion for Algorithm 1 is precisely that

$$K_i = I \text{ holds for each } i \in I \qquad (34)$$

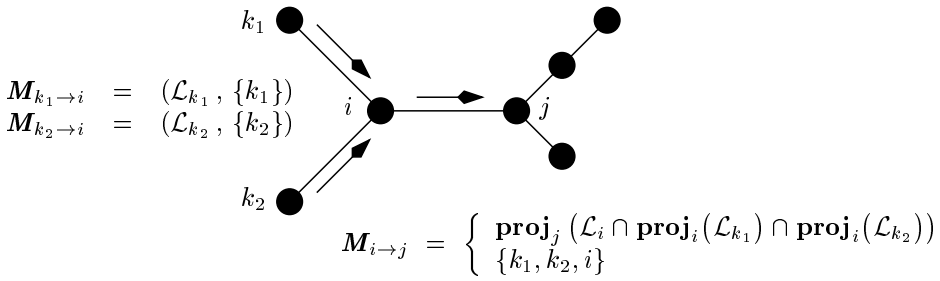

Figure 2: Illustrating step (30) of Algorithm 2, for a simplification of the system depicted in Fig. 1.

Now, (34) turns out to be an effective criterion for distributed termination if the Algorithm 2 with marks is used instead of the original Algorithm 1. Implementing (34), however, requires that each peer knows the set I of all sites. Albeit minimal, this is a global information about the system. It may require, *e.g.,* the implementation of a distributed protocol for *group membership* [43].

To summarize, if the set of all supervising peers has identified itself as a *group,* then Algorithm 2 with marks allows using (34) as an effective criterion for distributed termination. If supervisor i only knows its local model $\mathcal{L}_i$ and the interface $L_i \cap L_j$ with each of its neighbour j, then (34) cannot be effectively implemented.

Optimal scheduling. Fig. 3 shows an optimal scheduling of the different steps (30) of the message passing algorithm, for our tree-shaped system. This scheduling implements an inward seep followed by an outward sweep, where the thick node has been selected as a center for the tree. The steps having same index can be performed in any order, or even simultaneously. This scheme corresponds to the well known Rauch-Tung-Striebel two-sweep algorithm for linear systems smoothing [42]. This rigid scheduling minimizes the total number of messages exchanged by the sites. However, it requires global coordination and thus cannot be implemented by unsupervised peers attached to the different sites.

What happens if the interaction graph possesses cycles? Knowing that $\mathcal{G}_I$ is a tree is indeed a global information regarding the system. However, this is a milder information than actually knowing the graph, which is requested in order to apply termination criterion (34). Still, it makes sense investigating what happens when the interaction graph $\mathcal{G}_I$ possesses cycles. In this case, the same Algorithm 1 can still be used. At the equilibrium, it yields *local consistency* in the sense of [46, 47, 48], namely:

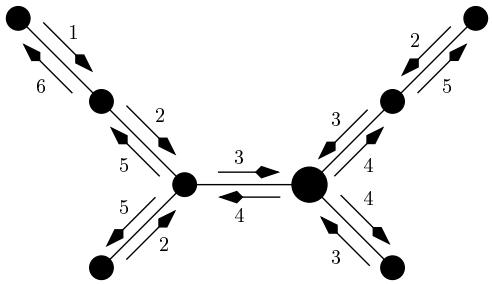

Figure 3: An optimal scheduling for the different steps of the message passing algorithm.

Theorem 4 *Without any assumption on $\mathcal{G}_I$, Algorithm 1 converges to $(\mathcal{L}_i^\infty)_{i\in I}$ satisfying*

$$\forall (i,j) \in \mathcal{G}_I \quad : \quad \mathbf{proj}_i(\mathcal{L}_j^\infty) = \mathbf{proj}_j(\mathcal{L}_i^\infty) \tag{35}$$

but $\mathcal{L}_i^\infty \neq \mathbf{proj}_i(\mathcal{L})$ in general.

Sketch of proof: The idea of the proof consists in showing that Algorithm 1 can be modified as follows, without changing its convergence behaviour. Instead of performing (26), perform

$$\mathcal{M}_{i\to j} \quad := \quad \mathbf{Msg}_{i\to j} \circ \mathbf{Fuse}\left[\mathcal{L}_i \times_L \omega_i\,,\, \{\,\mathcal{M}_{k\to i} \mid (k,i) \in \mathcal{G}_I, k \neq i\}\right] \tag{36}$$

Note that, unlike in (26), the present message $\mathcal{M}_{i\to j}$ bounces back the incoming message $\mathcal{M}_{j\to i}$. The fact that this is a legal modification of Algorithm 1 follows from the following property: for any two alphabets L and L', and $\mathcal{L}$ any language over L:

$$\mathcal{L} \times_L \mathbf{proj}_{L'}(\mathcal{L}) \quad = \quad \mathcal{L}, \tag{37}$$

which E. Fabre calls *involutivity* [17, 21]. Note that (37) is not a consequence of the properties collected in Lemma 1. It is thus an additional feature of languages, equipped with their projections and shuffle product. Having replaced (26) by (36) has the advantage that, now, $\mathcal{M}_{i\to j} \subseteq \mathcal{M}_{j\to i}$, whence equality follows, which is precisely (35). ⋄

We refer the reader to [17, 21] for an extensive discussion of this situation and the role of involutivity in the design of message passing algorithms, for various contexts.

3.5 Distributed modular on-line monitoring — Objective 2

So far we addressed Objective 1. In this section, we consider Objective 2, namely on-line monitoring. The situation is the following. In Section 3.4 we assumed that observation ω_i was globally available at site i before the distributed algorithm could proceed. This manifested itself by the fact that local observations ω_i keep constant throughtout the iteration steps of Algorithm 1.

On-line monitoring algorithm. In this section, we consider on-line monitoring algorithms. Referring to Algorithm 1, this means that each supervisor collects its local observation ω_i incrementally, in successive packets of alarms. The growing of observation ω_i at each site interleaves with the iteration steps of the message passing algorithm. We shall consider both terminating and non terminating observations.

To model the process of collecting observations, consider the following operator attached to site i, where ω_i is the sequence of observations stored by this site up to the current instant:

$$\mathbf{Grow}(\,\omega_i\,,\,o_i\,) \quad =_{\rm def} \quad \text{append to } \omega_i \text{ a new finite sequence } o_i \text{ of local alarms.} \tag{38}$$

The following new problem occurs, due to the on-line nature of the algorithm combined with its asynchrony. Consider the simple example of two supervisors collecting alarms from the same sensor, with alphabet $L = \{a, b\}$. Since communication is asynchronous, it may be that, at some instant, supervisor 1 has observed $aabba$, whereas the supervisor 2 has only observed aab (note that this cannot happen with the off-line monitoring problem, which assumes that all alarms have been collected before starting the algorithm). With these two observations, the two supervisors won't be in general able to agree on any explanation. This situation may prevail for ever, assuming that supervisor 1 is always quicker at getting observations than supervisor 2. The solution is clear: we must compensate for the possible delay in using local observations (and exchanging messages). This is achieved by being cautious: when receiving $aabba$, supervisor 1 will interpret this as $aabba.\{a,b\}^*$, whereas supervisor 2 will interpret aab as $aab.\{a,b\}^*$. But, now, $aabba.\{a,b\}^*$ and $aab.\{a,b\}^*$ have a non empty intersection, implying that the two supervisors will find a common set of agreable explanations. This amounts to interpreting the increase of observations as a decrease in the set of possible futures. Note that this is different from taking prefix closure.

Accordingly, for ω a word over alphabet L and $\mathcal{L}$ a language over L, define the *completions*

$$\overline{\omega} \;=_{\rm def}\; \omega.L^* \quad , \quad \overline{\mathcal{L}} \;=_{\rm def}\; \mathcal{L}.L^*. \tag{39}$$

Note that, if ω' is a prefix of ω, then $\overline{\omega'} \supseteq \overline{\omega}$. Using operator (38) and notation (39), the on-line version of Algorithm 1 is as follows (note the use of completions in steps 3 and 4):

Algorithm 3 (on-line message passing algorithm) *The different supervising peers, respectively attached to each site $i \in I$, perform, independently and in a chaotic way:*

- Initialization: *each peer $i \in I$ initializes its set of messages as follows:*

$$\forall (i,j) \in \mathcal{G}_I \quad : \quad \mathcal{M}_{i\to j} := (L_i \cap L_j)^* \tag{40}$$

- Chaotic iteration step: *each peer $i \in I$ performs one of the following two alternatives:*
  - Collect local observations: *perform $\omega_i := \mathbf{Grow}(\,\omega_i\,,\,o_i\,)$, where o_i are the newly collected observations, at peer i.*
  - Update and propagate messages: *using currently available local observations ω_i,*

1. *pick* $(i,j) \in \mathcal{G}_I$ *and define* $J_{ij} \subseteq \{k \in I \mid (i,k) \in \mathcal{G}_I, k \neq j\}$;
2. *read the current value of* $\mathcal{M}_{k\to i}$, *for* $k \in J_{ij}$;
3. *update message to peer* j:

$$\mathcal{M}_{i\to j} \quad := \quad \mathbf{Msg}_{\,i\to j} \circ \mathbf{Fuse}\,[\,\mathcal{L}_i \times_{_L} \overline{\omega}_i\,,\,\{\,\mathcal{M}_{k\to i} \mid k \in J_{ij}\}\,] \tag{41}$$

4. *read the current value of* $\mathcal{M}_{j\to i}$, *for* $(i,j) \in \mathcal{G}_I, j \neq i$ *and update*

$$\mathcal{L}'_i \quad := \quad \mathbf{Fuse}\,[\,\mathcal{L}_i \times_{_L} \overline{\omega}_i\,,\,\{\,\mathcal{M}_{j\to i} \mid (i,j) \in \mathcal{G}_I, j \neq i\}\,] \tag{42}$$

If collecting on-line observations is a non terminating process, then so is this algorithm. Thus "on-line" is performed in a non strict sense meaning that each supervisor decides at will when to exploit freshly received alarms from its sensor, so that there may be several of them (whence the definition for (38)).

Analysis of on-line monitoring Algorithm 3. For $\mathcal{L}$ and $\mathcal{L}'$ two languages over the same alphabet L, defined the following partial order relation:

$$\mathcal{L}' \preceq \mathcal{L} \quad \text{iff} \quad \overline{\mathcal{L}}' \subseteq \overline{\mathcal{L}}. \tag{43}$$

Note that $\mathcal{L}' \subseteq \mathcal{L}$ implies $\mathcal{L}' \preceq \mathcal{L}$ but the converse is not true, as shown by the case of $\mathcal{L}' = \{w'\}$ and $\mathcal{L} = \{w\}$, where w is a prefix of w'. Partial order $\preceq$ extends to languages defined over different alphabets as usual, by taking inverse projections to equalize their alphabets. Using the $\preceq$ order, Lemma 2 refines as follows:

Lemma 3 *The two maps*

$$\begin{array}{rcl} \mathcal{V}_i & \mapsto & \mathbf{Msg}_{\,i\to j}\,(\mathcal{V}_i) \\ (\mathcal{V}_i, \mathcal{V}'_i) & \mapsto & \mathbf{Fuse}\,[\,\mathcal{V}_i, \mathcal{V}'_i\,] \end{array}$$

where $j \in I$ *is arbitrary, are increasing w.r.t. all their arguments, for the order* $\preceq$ *defined in (43). Furthermore,* $\mathbf{Fuse}\,[\,\mathcal{V}_i, \mathcal{V}'_i\,] \preceq \mathcal{V}_i$, *and, for any pair* (ω_i, o_i), *we have*

$$\mathbf{Grow}(\,\omega_i\,,\,o_i\,) \quad \preceq \quad \omega_i$$

The following result characterizes the behaviour of Algorithm 3. Note that it applies to a possibly non terminating algorithm.

Theorem 5 *On-line chaotic message passing Algorithm 3 converges in the following sense:*

1. *The following property is maintained by this algorithm:*

$$\mathcal{L}'_i \quad \succeq \quad \mathbf{proj}_i\big(\mathcal{L} \times_{_L} \Omega\big),\ \textit{where}\ \Omega = \textstyle\bigtimes_{i\in I}^{L} \omega_i, \tag{44}$$

where $\mathcal{L}'_i$ *is the current solution computed by peer* i, *and* ω_i *is the current observation collected on-line at peer* i.

2. *If the algorithm is fairly executed in the sense of Definition 3, then the sequence of successive updates of $\mathcal{L}'_i$ is decreasing and eventually satisfies:*

$$\mathcal{L}'_i \quad \preceq \quad \mathbf{proj}_i\big(\mathcal{L} \times_L \Omega_{\rm stop}\big) \tag{45}$$

where $\Omega_{\rm stop}$ is any fixed *prefix of the growing observation Ω.*

3. *If observation Ω has bounded cardinality, and if the algorithm is fairly executed, then $\mathcal{L}'_i$ eventually converges to the desired solution $\mathbf{proj}_i\big(\mathcal{L} \times_L \Omega\big)$.*

Properties (44) and (45) characterize the kind of convergence the on-line chaotic Algorithm 3 satisfies. Property (44) expresses that the on-line message passing algorithm provides all correct solutions to the monitoring problem as well as possibly additional spurious solutions; the reason for this is that, being unsupervised and asynchronous, the algorithm may be "late" at processing either recent observations or recent messages from the other supervisors. On the other hand, property (45) expresses that, eventually, the algorithm will correctly explain every fixed prefix of the observations. The corresponding delay is finite but not bounded (unless quantitative assumptions are made on the duration of communications). Last but not least, we insist that completions are only used for the purpose of the analysis, not in the algorithms themselves.

Proof: We use the same technique as for the proof of the off-line algorithm, by enhancing the on-line version with additional marks used only in the proof. The marks will be more involved to account for the asynchrony in getting observations. In the off-line algorithm, the mark K_{ij} attached to message $\mathcal{M}_{i\to j}$ indicated the set of sites taken into account by this message. For the on-line algorithm, the information carried over each site k by message $\mathcal{M}_{i\to j}$ will have a "date", corresponding to the length of the observation at site k used in message $\mathcal{M}_{i\to j}$.

Formally, for $i \in I$, let ω_i^∞ be the entire observation at peer i — it can be either finite or infinite. This observation is the concatenation of a finite or infinite sequence of successive finite blocks of alarms: $\omega_i^\infty = o_i(1).o_i(2).o_i(3)\ldots$ The observation collected by peer i at an arbitrary instant of the on-line algorithm is generically denoted by ω_i; it is a prefix of ω_i^∞ and has the form $\omega_i = o_i(1).o_i(2).o_i(3)\ldots o_i(n)$, for some finite index n equal to the number of successive reads performed by the site. For ω_i as above and m an integer, let ω_i/m be the observation ω_i truncated at m, equal to

$$\omega_i/m \quad =_{\rm def} \quad o_i(1).o_i(2).o_i(3)\ldots o_i(m) \text{ if } m \le n, \text{ and } \omega_i \text{ otherwise.} \tag{46}$$

The messages will thus be marked as follows:

$$\boldsymbol{M}_{i\to j} \quad =_{\rm def} \quad (\mathcal{M}_{i\to j}, \tau_{ij})$$

where τ_{ij} is a *mark*, *i.e.,* an element

$$\tau \quad \in \quad \mathbb{N}^I$$

where $\mathbb{N} = 0, 1, 2, 3 \ldots$ is the set of nonnegative integers; $\tau_{ij}(k) = m$ indicates that ω_k/m has been taken into account in message $\mathcal{M}_{i\to j}$. For $\tau \in \mathbb{N}^I$ a mark as above,

$$\text{let } K_\tau \text{ be the set of } i \in I \text{ such that } \tau(i) > 0, \tag{47}$$

i.e., K_τ is the support of τ. For $\omega_i = o_i(1).o_i(2).o_i(3) \ldots o_i(m)$ a finite local observation of length m,

$$\text{let } \tau_{\omega_i} \text{ be the mark such that } \tau_{\omega_i}(i) = m \text{ and } \tau_{\omega_i}(k) = 0 \text{ for } k \neq m.$$

Finally, the operators are enhanced as follows:

$$\begin{array}{rcl} \boldsymbol{Msg}_{i\to j}\,(\mathcal{L}, \tau) & =_{\text{def}} & \text{send to site } j \text{ the pair } : \big(\mathbf{proj}_j(\mathcal{L}), \tau\big) \\ \boldsymbol{Fuse}\,[\,(\mathcal{L}, \tau), (\mathcal{L}', \tau')\,] & =_{\text{def}} & \big(\mathbf{Fuse}\,[\,\mathcal{L}, \mathcal{L}'\,],\ \tau \vee \tau'\big) \end{array}$$

where supremums are taken componentwise. With these notations, we are now ready to state our enhanced on-line algorithm:

Algorithm 4 (chaotic on-line message passing, with marks) *The different supervising peers, respectively attached to each site $i \in I$, perform, independently and in a chaotic way:*

- Initialization: *each peer $i \in I$ initializes its set of messages as follows:*

$$\forall (i,j) \in \mathcal{G}_I \quad : \quad \boldsymbol{M}_{i\to j} := ((L_i \cap L_j)^*, 0) \tag{48}$$

- Chaotic iteration step: *each peer $i \in I$ performs one of the following two alternatives:*
  - Collect local observations: *perform $\omega_i := \mathbf{Grow}(\,\omega_i\,,\, o_i\,)$, where o_i are the fresh observations collected at peer i;*
  - Update and propagate messages:
    1. *pick $(i,j) \in \mathcal{G}_I$ and define $J_{ij} \subseteq \{k \in I \mid (i,k) \in \mathcal{G}_I, k \neq j\}$;*
    2. *read the current value of $\boldsymbol{M}_{k\to i}$, for $k \in J_{ij}$;*
    3. *update message to peer j:*

$$\boldsymbol{M}_{i\to j} \quad := \quad \boldsymbol{Msg}_{i\to j} \circ \boldsymbol{Fuse}\,[\,(\mathcal{L}_i \times_L \overline{\omega}_i, \tau_{\omega_i})\,,\, \{\,\boldsymbol{M}_{k\to i} \mid k \in J_{ij}\}\,]$$

    4. *read the current value of $\boldsymbol{M}_{j\to i}$, for $(i,j) \in \mathcal{G}_I, j \neq i$ and update*

$$(\mathcal{L}'_i, \tau_i) \quad := \quad \boldsymbol{Fuse}\,[\,(\mathcal{L}_i \times_L \overline{\omega}_i, \tau_{\omega_i})\,,\, \{\,\boldsymbol{M}_{j\to i} \mid (i,j) \in \mathcal{G}_I, j \neq i\}\,] \tag{49}$$

By reasoning as in (32), we prove that the following invariant is maintained by Algorithm 4: for each branch $(i,j) \in \mathcal{G}_I$:

$$\boldsymbol{M}_{i\to j} = (\mathcal{M}_{i\to j}, \tau_{ij}) \quad \Rightarrow \quad \mathcal{M}_{i\to j} \;=\; \mathbf{proj}_{ij}\left(\times^L_{k\in K_{\tau_{ij}}} \left(\mathcal{L}_k \times_L \overline{\omega_k/\tau_{ij}(k)}\,\right)\right), \tag{50}$$

where we recall that $K_{\tau_{ij}}$ is the support of τ_{ij} (see (47)), and $\mathbf{proj}_{ij} =_{\text{def}} \mathbf{proj}_{L_i \cap L_j}$. Invariant (50) implies the following two weaker invariants:

$$\mathcal{L}'_i \;\succeq\; \mathbf{proj}_i\big(\bigtimes^L_{k\in I} (\mathcal{L}_k \times_L \overline{\omega}_k)\big) \tag{51}$$

$$\forall m \geq 1 \;:\; \mathcal{L}'_i \;\preceq\; \mathbf{proj}_{ij}\left(\bigtimes^L_{k\in K_{\tau_{ij}}} \left(\mathcal{L}_k \times_L \overline{\omega_k/m'}\right)\right), \tag{52}$$

where $m' = \min(\tau_{ij}(k), m)$. Invariant (51) is statement 1 of the theorem, whereas invariant (52) proves its statement 2. Finally, statement 3 is a direct consequence of invariant (50). This finishes the proof of Theorem 5. ◇

3.6 Back to distributed monitoring in terms of runs

In this section we briefly explain how to extend the methods of sections 3.2 – 3.5 to handle monitoring in terms of runs as defined in Section 2. Central to our previous message passing algorithms was the homogeneous nature of the objects involved, namely languages. Languages were used to express both the distributed system, its observations, and the solution to the monitoring problem.

The problem with the monitoring as defined in Section 2 is that it involves a mix of languages (for the observations) and of runs (to express the solutions of the monitoring problem). Whereas runs cannot be reduced to languages (since states are involved), languages can be lifted to sets of runs, as we explain next. A run can be seen as a special kind of automaton, consisting of a chain alternating labeled states and labeled transitions.

Chains and chain processes. More precisely, call a *chain* any word alternating symbols from two finite alphabets S and L, *i.e.,* an element of $(SL)^*S$. Run σ of formula (1) can be seen as a chain; hence, chains will be represented by means of notation (1). Call (S, L)-*chain process*, or simply *chain process* if no confusion can occur, any sub-language of $(SL)^*S$. Chain processes are generically denoted by the symbol Σ. They represent sets of runs of automata in a flat, unstructured, manner; very much like languages do for observations.

Basic operations. Chain processes are equipped with the following operations.

- For Σ and Σ' two (S, L)-chain processes, their *intersection* $\Sigma \cap \Sigma'$ is the set of common chains to Σ and Σ'.
- For σ an (S, L)-chain, $L' \subseteq L$ and $\pi : S \mapsto S'$ a total surjection from S onto some alphabet S', let

  $$\mathbf{proj}_{L,L';\pi}(\sigma) \tag{53}$$

  be the *projection* of σ onto L' along π, obtained by applying the following two rules to chain σ, where the term "maximal" refers to partial ordering by inclusion:

1. any maximal sub-chain

$$s_k \xrightarrow{\ell_{k+1}} s_{k+1} \xrightarrow{\ell_{k+2}} s_{k+2} \xrightarrow{\ell_{k+3}} s_{k+3} \ldots s_{k+n-1} \xrightarrow{\ell_{k+n}} s_{k+n}$$

such that $\forall m = 1, \ldots, n-1, \ell_{k+m} \notin L'$ and $\ell_{k+n} \in L'$, is replaced by

$$\pi(s_k) \xrightarrow{\ell_{k+n}} \pi(s_{k+n});$$

2. any maximal sub-chain

$$s_k \xrightarrow{\ell_{k+1}} s_{k+1} \xrightarrow{\ell_{k+2}} s_{k+2} \xrightarrow{\ell_{k+3}} s_{k+3} \ldots s_{k+n-1} \xrightarrow{\ell_{k+n}} s_{k+n}$$

such that $\forall m = 1, \ldots, n, \ell_m \notin L'$, is replaced by $\pi(s_k)$.

States that are not connected in the resulting chain are removed. For Σ an (S, L)-chain process, $\mathbf{proj}_{L,L';\pi}(\Sigma) = \{\mathbf{proj}_{L,L';\pi}(\sigma) \mid \sigma \in \Sigma\}$ is the *projection* of Σ onto L' along π. We simply write $\mathbf{proj}_{L';\pi}(\Sigma)$ when no confusion can result. Finally, when $L' = L$ we omit these alphabets and write

$$\mathbf{proj}_{\pi}(\Sigma) \text{ instead of } \mathbf{proj}_{L,L;\pi}(\Sigma). \tag{54}$$

Please, note that the above projection operation is different from applying the usual projections for languages to σ, seen as a word of $(SL)^*S$.

- Finally, for $i = 1, 2$, let Σ_i be an (S_i, L_i)-chain process. The *product* $\Sigma_1 \times_C \Sigma_2$ is defined by

$$\Sigma_1 \times_C \Sigma_2 \quad = \quad \mathbf{proj}^{-1}_{L,L_1;\pi_1}(\Sigma_1) \cap \mathbf{proj}^{-1}_{L,L_2;\pi_2}(\Sigma_2)$$

where $L = L_1 \cup L_2$, $S = S_1 \times S_2$, and π_i is the projection from S onto S_i, for $i = 1, 2$.

An effective algorithm for computing this product is proposed in Appendix A.1.

Distributed monitoring. Problem 1 is reformulated in terms of chain processes as follows. Let $\mathcal{A} = (S, L, \rightarrow, S_0)$, where $L = L_o \uplus L_u$, be the system for monitoring. The set $\Sigma_{\mathcal{A}}$ of all runs of $\mathcal{A}$ is an (S, L)-chain process. Regard an observation $\ell_1, \ell_2, \ldots, \ell_n \in \mathcal{L}_{\mathcal{A},o}$ as a $(\mathbb{N}, L_o)$-chain as follows:

$$\omega \quad = \quad 0 \xrightarrow{\ell_1} 1 \xrightarrow{\ell_2} 2 \ldots n-1 \xrightarrow{\ell_n} n$$

where the integers $0, 1, 2, \ldots, n$ label the nodes of the chain. These integers count the length of the observation. A set of observations Ω is thus an $(\mathbb{N}, L_o)$-chain process.

Definition 4 *Using notational convention (54), the monitor of $\mathcal{A}$ in terms of runs is the map*

$$\Omega \quad \longmapsto \quad \mathcal{M} =_{\rm def} \mathbf{proj}_\pi\big(\Sigma_{\mathcal{A}} \times_C \Omega\big), \tag{55}$$

where $\pi : S \times \mathbb{N} \mapsto S$ is the projection over the set of states of the system for monitoring.

The projection $\mathbf{proj}_\pi$ erases the component of the state originating from the observation and otherwise has no effect — note that this was not needed when performing monitoring in terms of languages. The language $\mathcal{L}_{\mathcal{M}}$ generated by monitor (55) coincides with the result provided by monitor (9).

Definition 4 also applies if $\mathcal{A} = \times_{i\in I}\mathcal{A}_i$ and $\Omega = \times^C_{i\in I}\omega_i$, with the slight difference that Ω is now an $(\mathbb{N}^I, L_o)$-chain process. We leave the reader as an (easy) exercise to reformulate and prove the counterpart of Theorem 1 and Lemma 1, for chain processes. The *completion* of (S, L)-chain process Σ is defined by

$$\overline{\Sigma} \quad =_{\rm def} \quad \Sigma.(SL)^*S, \tag{56}$$

and the order relation $\Sigma' \preceq \Sigma$ is defined by $\overline{\Sigma'} \subseteq \overline{\Sigma}$. Again, Lemmas 2 and 3 are easily extended to chain processes.

Since Theorem 1 and Lemmas 1–3 were the only foundations needed to develop modular monitoring with its distributed message passing algorithms, both off-line and on-line, the latter carry over to chain processes, which solves Problem 1 of distributed monitoring in terms of runs.

4 Efficient data structures

4.1 Motivation — Objective 3

So far we have represented the set of solutions to the monitoring problem as a flat, unstructured, set of chains. Our message passing algorithms will manipulate the same kind of data structures. Since distributed systems involves lots of concurrency, this quickly becomes intractable, especially for on-line algorithms. So the following central (informal) requirement emerges:

Requirement 1 (addressing Objective 3) *Represent the set of runs $\Sigma_{\mathcal{A}}$ of an automaton $\mathcal{A}$ in the form of some efficient data structure $\mathcal{D}_{\mathcal{A}}$, with the following features:*

1. *A notion of intersection can be defined over this data structure, which parallels the intersection of chain processes in that $\mathcal{D} \cap \mathcal{D}'$ represents $\Sigma \cap \Sigma'$, for Σ and Σ' sets of runs defined over the same alphabet. $\mathcal{D} \cap \mathcal{D}'$ can be computed directly on $\mathcal{D}$ and $\mathcal{D}'$, without the need to unwrap them back to Σ and Σ'.*
2. *The projection $\mathbf{proj}_{L,L_o;\pi}\big(\mathcal{D}_{\mathcal{A}}\big)$ can be directly computed on $\mathcal{D}_{\mathcal{A}}$, without the need to unwrap it back to $\Sigma_{\mathcal{A}}$.*

3. *A product can be defined for this data structure, such that* $\mathcal{D}_{\mathcal{A}\times\mathcal{A}'} = \mathcal{D}_{\mathcal{A}} \times \mathcal{D}_{\mathcal{A}'}$*, and this product can be computed directly on* $\mathcal{D}_{\mathcal{A}}$ *and* $\mathcal{D}_{\mathcal{A}'}$*, without the need to unwrap them back to* $\Sigma_{\mathcal{A}}$ *and* $\Sigma_{\mathcal{A}'}$*.*

4. *The above operations satisfy Theorem 1, Lemma 1, and support partial order* $\preceq$ *introduced in (43) for the analysis of on-line algorithms.*

The rest of the paper investigates various means to satisfy Requirement 1, with increasing efficiency.

In a first stage, we shall investigate how to store and manipulate runs of automata efficiently. In section 3.6, we consistently represented sets of runs in a totally flat manner. This is clearly stupid and a first improvement consists in taking into account that runs are partially ordered by the prefix order, with a unique minimum consisting of the empty run. This trivial remark leads to representing the executions of an automaton as its "execution tree", where partial runs are represented only once. Developing distributed monitoring techniques using execution trees is investigated in Section 4.2.

Now, a number of algorithms from control engineering and optimization manipulate sets of executions of automata. Examples include dynamic programming and the Viterbi algorithm. Those algorithms represent sets of executions in the form of "trellises", in which runs of identical length ending at the same state are merged (the rationale being that they share the same future). We shall devote the longer section 4.3 to distributed monitoring techniques using trellises. As the reader will notice, this subject is full of pitfalls and must be investigated with extreme care. In particular, meeting Requirement 1.4 is non trivial. A surprising conclusion will be that making trellises suitable for distributed processing will bring us very close to partial orders.

Thus the last step naturally consists in further increasing the efficiency of data structures by taking concurrency into account, *i.e.,* the fact that independent and unrelated moves in a system need not be represented through their many possible interleavings, but only once by means of a partial order. This is the subject of Section 5.

4.2 Execution trees

4.2.1 Definition

Execution trees consist in representing sets of runs by superimposing common prefixes of the latter, thus obtaining a tree-shaped data structure. Formally, an (S, L)-*execution tree* is a tree whose branches and nodes are labeled by two finite alphabets denoted by L and S. We do not distinguish execution trees that are related by a label preserving bijection between their nodes and branches, respectively.

For $\mathcal{A} = (S, L, \rightarrow, s_0)$ an automaton, let $\mathcal{U}_{\mathcal{A}}$ denote the (unique) maximal (S, L)-execution tree whose all branches are maximal runs of $\mathcal{A}$, each such run being represented only once. Call $\mathcal{U}_{\mathcal{A}}$ the *execution tree* of $\mathcal{A}$. Fig. 4 shows an automaton and a prefix of its execution tree (execution trees are infinite as soon as automata possess loops).

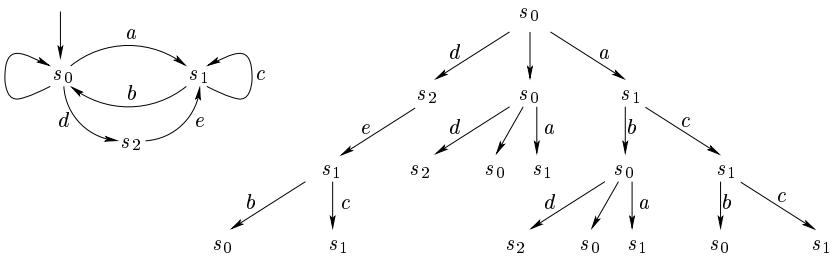

Figure 4: Automaton $\mathcal{A}$ and a prefix of its execution tree $\mathcal{U}_{\mathcal{A}}$. (The reader is kindly invited to draw the corresponding chain process, for comparison; in doing this, please, remember that the different prefixes of a same chain must be listed.)

Execution trees are clearly a much more compact data structure than chain processes. However, they raise the following new issue: an (S, L)-execution tree $\mathcal{V}$, as defined above, can only represent a *prefix closed* language. Whereas this is fine when representing the sets of all runs of an automaton, it is no longer convenient to capture observations, which are *not* prefix closed. Languages that are not prefix closed can be represented as execution trees equipped with an extra boolean marking, to indicate the allowed final states. Formally:

Definition 5 (execution tree) *An* (S, L)-execution tree *is a triple* $\mathcal{V} = (\mathbf{T}, \lambda, f)$*, where* $\mathbf{T}$ *is a tree,* λ *is a labeling of the tree, mapping nodes to* S *and branches to* L*, and* f : *nodes*$(\mathbf{T}) \mapsto \{0, 1\}$*, is the* stop function*. Call* stop point *any node mapped to* 1 *by* f *and call* run *any branch of* $\mathbf{T}$ *that is either infinite or ending at a stop point. Whenever convenient, we shall denote the components of* $\mathcal{V}$ *by* $\mathbf{T}_{\mathcal{V}}, \lambda_{\mathcal{V}}$*, and* $f_{\mathcal{V}}$*.*

The correspondence between execution trees and chain processes is as follows:

Lemma 4 *Each* (S, L)*-execution tree* $\mathcal{V} = (\mathbf{T}, \lambda, f)$ *gives raise to a unique chain process* $\Sigma_{\mathcal{V}}$ *having identical sets of runs, and vice-versa. We denote by* Φ *this one-to-one correspondence, from execution trees to chain processes.*

In $\mathcal{U}_{\mathcal{A}}$, the execution tree of an automaton $\mathcal{A}$, every node is a stop point. Non trivial stop functions are necessary to represent sets of observations as execution trees.

4.2.2 Operations on execution trees

To be able to express monitoring in terms of execution trees, we need to equip them with operations as described in Requirement 1. These are introduced next:

- For $\mathcal{V}$ and $\mathcal{V}'$ two (S, L)-execution trees, their *intersection* is defined by

$$\mathcal{V} \cap \mathcal{V}' \quad =_{\text{def}} \quad \Phi^{-1}(\Phi(\mathcal{V}) \cap \Phi(\mathcal{V}'))$$

- Let $\mathcal{V}$ be an (S, L)-execution tree. For $L' \subseteq L$ and $\pi : S \mapsto S'$ a total surjection from S onto some alphabet S', the *projection* of $\mathcal{V}$ onto L' along π is defined by:

$$\mathbf{proj}_{L,L';\pi}(\mathcal{V}), \text{ or, simply } \mathbf{proj}_{L';\pi}(\mathcal{V}) \quad =_{\text{def}} \quad \Phi^{-1}\left(\mathbf{proj}_{L,L';\pi}(\Phi(\mathcal{V}))\right) \tag{57}$$

- The *product* of execution trees is defined as follows:

$$\mathcal{V} \times_U \mathcal{V}' \quad =_{\text{def}} \quad \Phi^{-1}\left(\Phi(\mathcal{V}) \times_C \Phi(\mathcal{V}')\right) \tag{58}$$

  When $\mathcal{V} = \mathcal{V}' \times_U \mathcal{V}''$, we simply write

$$\mathbf{proj}_{\mathcal{V}'}(\mathcal{V}) \quad \text{instead of} \quad \mathbf{proj}_{L,L';\pi}(\mathcal{V}) \tag{59}$$

While the above definitions are mathematically convenient, they are not effective and do not satisfy the requirement that these operations can be performed directly on the data structures themselves, without unwrapping them back to chain processes. The following result is therefore essential:

Theorem 6 *The above operations of intersection, projection, and product, can be computed directly on execution trees.*

Proof: See Algorithm 6 in Appendix A.2. ◇

4.2.3 Execution tree based monitoring

The monitor for $\mathcal{A} = (S, L, \rightarrow, s_0), L = L_o \cup L_u$ is redefined in terms of execution trees as follows. We first need to represent observations as execution trees. To this end, note that local observations ω_i can be represented as an $(\mathbb{N}, L_{o,i})$-execution tree with a single branch. Thus we can represent the global observations by the $(\mathbb{N}^I, L_o)$-execution tree

$$\Omega \quad = \quad \times^U_{i \in I} \omega_i,$$

and, using notational convention (54), the global monitor is simply defined by the map

$$\Omega \quad \longmapsto \quad \mathcal{M} \ =_{\text{def}} \ \mathbf{proj}_{\pi}\left(\mathcal{U}_{\mathcal{A}} \times_U \Omega\right), \tag{60}$$

where $\pi : S \times \mathbb{N}^I \mapsto S$ is the projection over the set of states of the system for monitoring (the component of the state originating from the observation is erased).

This is illustrated in Fig. 5. The construction of $\mathcal{M}$ can be performed incrementally and on-line, while successive events of Ω are received. Note that, when the second observation is being processed, the branch $(s_1, 0) \xrightarrow{f} (s_3, 1)$ offers no continuation to explain the postfix $\{r, f\}$ of the observation sequence; it must therefore be pruned out from $\mathcal{M}$. Such a pruning can be performed with delay exactly 1, *i.e.,* on reception of the event labeled r in Ω.

Now, since, by Section 4.2.2, Φ mirrors the basic operations on chain processes with the corresponding ones on execution trees, the apparatus composed of Theorem 1, partial order

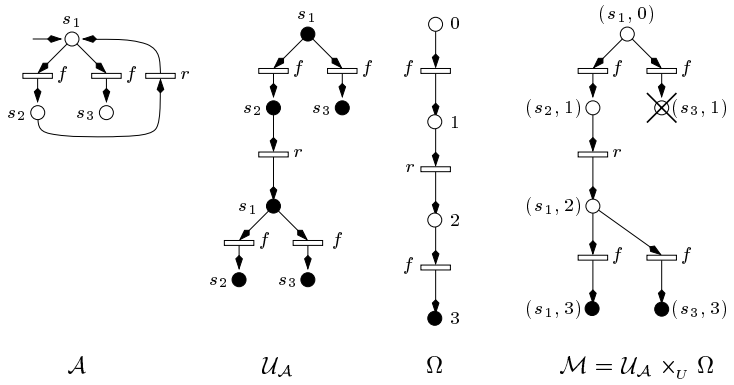

Figure 5: Computing the monitor $\mathcal{U}_\mathcal{A} \times_U \Omega$. Stop points of execution trees are marked in black. The branch whose node is crossed does not belong to the product and must be pruned out.

$\preceq$, and Lemmas 1–3 carries over to execution trees. Having this at hand, we can consider *modular monitoring* with execution trees, defined as the map:

$$(\omega_i)_{i\in I} \quad \longmapsto \quad \mathcal{M}_{\rm mod} =_{\rm def} \left(\mathbf{proj}_{L_i;\pi_i}(\mathcal{U}_\mathcal{A} \times_U \Omega) \right)_{i\in I}, \tag{61}$$

where, for each i, $\Omega = \bigtimes^U_{i\in I} \omega_i$, ω_i is an observation for $\mathcal{A}_i$, and $\pi_i : S \times \mathbb{N}^I \mapsto S_i$ is the projection over the ith local state of the system for monitoring. Algorithm 1 as well as its on-line version Algorithm 3 carry over to execution trees. We insist that, while performing the steps of these algorithms, we never need to unwrap execution trees back to a flat set of runs.

Discussion. Did we address Requirement 1 properly? Not quite so: our solution is somehow cheating. In general, execution trees grow exponentially in width with their length, see Fig. 4. This becomes particularly prohibitive when considering on-line algorithms. We would be happy with data structures having bounded width along the processing. Trellises, which have been used for a long time in dynamic programming algorithms, are good candidates for this. The next section is devoted to this more compact data structure.

4.3 Trellises

Execution trees are a simple structure to represent sets of runs, for automata. However, when a path of the execution tree branches, its descendants separate for ever. To overcome this drawback, *trellises* have been used in dynamic programming (or in the popular Viterbi algorithm), by merging, in the execution tree, futures of different runs according to appropriate criteria.

4.3.1 Observation criteria

For example, we may consider merging terminal nodes of two finite runs σ and σ' if they satisfy the following two conditions:

1. They begin and terminate at identical states (this first condition is mandatory to ensure that σ and σ' have identical futures);
2. They are equivalent according to one of the following *observation criteria*:
   (a) σ and σ' possess identical length;[1]
   (b) σ and σ' possess identical visible length (by not counting silent transitions);
   (c) Select some $L_o \subset L$, and require that σ and σ' satisfy $\mathbf{proj}_{L_o}(w_\sigma) = \mathbf{proj}_{L_o}(w_{\sigma'})$, where w_σ and $w_{\sigma'}$ are the words over L generated by runs σ and σ', and $\mathbf{proj}_{L_o}(.)$ denotes the projection of languages.
   (d) Assume $\mathcal{A} = \times_{i\in I}\mathcal{A}_i$ and require that σ and σ' have identical lengths when restricted to the different local alphabets L_i.

We now formalize the concept of observation criterion. In the following we will need to mark that a given transition is "silent", *i.e.,* has no label. This will be indicated by using an extra symbol "-".

Definition 6 (observation criterion) *An* observation criterion $\theta : L \cup \{\text{-}\} \mapsto \mathcal{L}_\theta$ *is a partial function mapping alphabet* $L \cup \{\text{-}\}$ *to some free monoid* $(\mathcal{L}_\theta, .)$. *For* $\ell \in L \cup \{\text{-}\}$, *write* $\theta(\ell) = \perp$ *to mean that* $\theta(\ell)$ *is undefined. For* $w \in L^*$, *we define recursively* $\theta(w\ell) = \theta(w).\theta(\ell)$, *and we take the convention that* $\perp^* = \epsilon$, *the empty subset of* $\mathcal{L}_\theta$.

Let $\mathcal{T}$ be a directed graph whose nodes are labeled by S and branches are labeled by $L\cup\{\text{-}\}$. For θ an observation criterion, say that two paths

$$s_{\text{init}} \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} s_2 \xrightarrow{\ell_3} s_3 \ldots s_{n-1} \xrightarrow{\ell_n} s_{\text{end}}$$

and

$$s'_{\text{init}} \xrightarrow{\ell'_1} s'_1 \xrightarrow{\ell'_2} s'_2 \xrightarrow{\ell'_3} s'_3 \ldots s'_{m-1} \xrightarrow{\ell'_m} s'_{\text{end}}$$

of $\mathcal{T}$ are θ-*equivalent* iff

$$\begin{array}{l} s_{\text{init}} = s'_{\text{init}}, s_{\text{end}} = s'_{\text{end}}, \text{ and} \\ \theta(\ell_1\ell_2\ell_3\ldots\ell_n) = \theta(\ell'_1\ell'_2\ell'_3\ldots\ell'_m) \end{array} \tag{62}$$

Note that, in the above definition, labels ℓ_i or ℓ'_j may be equal to "-".

[1] This is the observation criterion used in dynamic programming or Viterbi algorithm.

Notation. By abuse of notation, we shall sometimes write $\theta(w)$ instead of $\theta(\ell_1\ell_2\ell_3\ldots\ell_n)$, when $w=\ell_1\ell_2\ell_3\ldots\ell_n$ is the word produced by the above run.

Definition 7 (trellis) *An* (S,L,θ)-trellis *is a tuple* $\mathcal{T}=(\mathbf{G},\lambda,f,\theta)$*, where*

- $\mathbf{G}$ *is a directed graph,*
- λ *is a labeling of the graph, mapping nodes to* S *and paths to* L*,*
- $f:$ *nodes*$(\mathbf{G})\mapsto\{0,1\}$*, is the* stop function,
- $\theta: L\cup\{\text{-}\}\mapsto\mathcal{L}_\theta$ *is an observation criterion,*

satisfying the following condition: any two paths originate from the same node of $\mathcal{T}$ *and terminate at the same node of* $\mathcal{T}$ *iff they are* θ*-equivalent. Call* stop point *any node mapped to* 1 *by* f *and call* run *any path of* $\mathbf{G}$ *that is either infinite or ending at a stop point. Whenever convenient, we shall denote the components of* $\mathcal{T}$ *by* $\mathbf{G}_\mathcal{T}$*, etc.*

Note that the directed graph $\mathbf{G}$ may contain circuits and is therefore not a DAG. This contrasts with the classical notion of trellis used in dynamic programming and the Viterbi algorithm. Still, as a consequence of the definition, every circuit of $\mathbf{G}$ must be labeled by a word whose image by θ is ϵ. Observation criteria corresponding to the above examples (a)–(d) are:

(a) $\mathcal{L}_\theta=\{1\}^*$, and, $\forall\ell\in L\cup\{\text{-}\}$, $\theta(\ell)=1$, otherwise $\theta(\ell)=\perp$.

(b) $\mathcal{L}_\theta=\{1\}^*$, and, $\forall\ell\in L$, $\theta(\ell)=1$, otherwise $\theta(\ell)=\perp$.

(c) $\mathcal{L}_\theta=L_o^*$, and $\theta(\ell)=\ell$ iff $\ell\in L_o$, otherwise $\theta(\ell)=\perp$.

(d) $\mathcal{L}_\theta=\left(\{1\}^*\right)^I$ equipped with per-chain concatenation, and $\theta(\ell)(i)=1$ if $\ell\in L_i$, otherwise $\theta(\ell)=\perp$.

Trellises are illustrated in Fig. 6, for the above cases (a), (b), and (c). Case (d) will be discussed later. Trellises will be generically denoted by symbols $\mathcal{T}$ or $\mathcal{S}$ in the sequel.

At this point, we need to state the counterpart of Lemma 4 on the correspondence between execution trees and trellises. However, the situation is more involved, as not every execution tree can give raise to a trellis, even when equipped with an observation criterion.

Lemma 5 *Each* (S,L,θ)*-trellis* $\mathcal{T}$ *gives raise to a unique* (S,L)*-execution tree* $\mathcal{V}$ *having identical sets of runs. Conversely, for any observation criterion* θ*, each* (S,L)*-execution tree* $\mathcal{V}$ *satisfying the following condition:*

two θ*-equivalent paths are followed by isomorphic child* (S,L)*-execution trees,* (63)

gives raise to a unique (S,L,θ)*-trellis having identical sets of runs. We denote by* Ψ_θ *this one-to-one correspondence, from trellises to execution trees satisfying Condition (63). Note that* Ψ_θ *is parameterized by* θ*.*

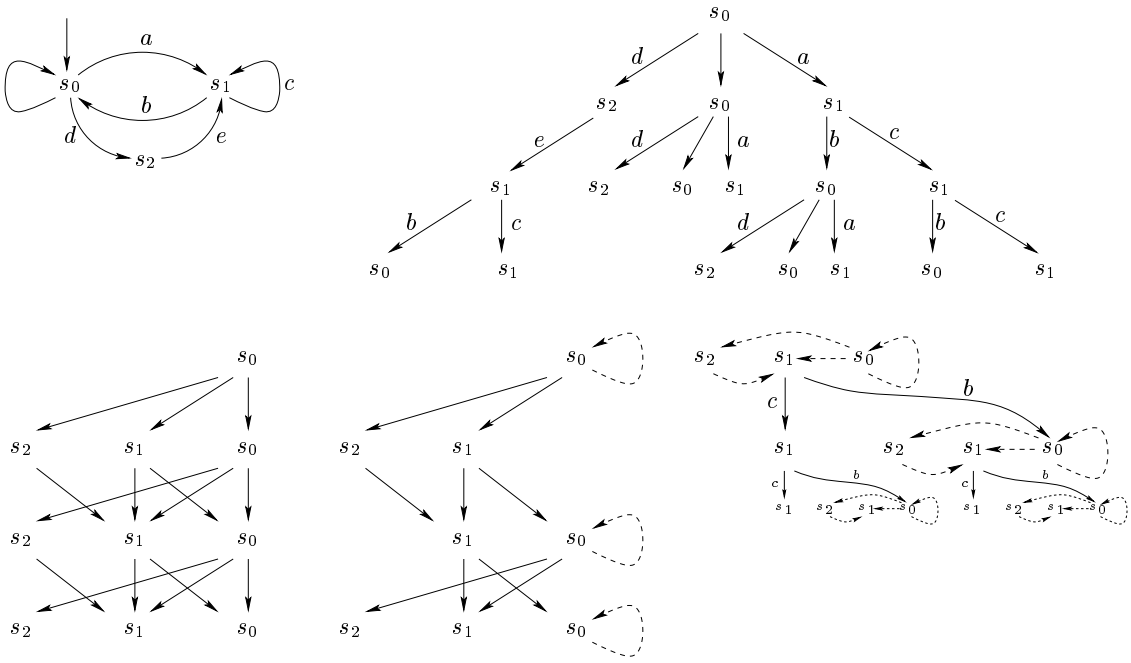

Figure 6: Top. Left: $\mathcal{A}$; right: execution tree $\mathcal{U}_A$. Bottom. Left: $\mathcal{T}_{\mathcal{A}}^{(a)}$; mid: $\mathcal{T}_{\mathcal{A}}^{(b)}$; right: $\mathcal{T}_{\mathcal{A}}^{(c)}$, with $L_o = \{b, c\}$. Labels of transitions are omitted in the trellises. Loops in trellises are dashed, they correspond to paths in the execution tree whose labels are undefined under observation criterion θ.

Proof: For the direct statement, just take the set of all runs of $\mathcal{T}$ and take for $\mathcal{V}$ the unique execution tree having this set of runs. Note that $\mathcal{V}$ satisfies Condition (63). For the converse part, if σ and σ' are two θ-equivalent paths of execution tree $\mathcal{V}$, then their respective terminal nodes possess isomorphic child trees, by Condition (63). Therefore, merging these two terminal nodes and their respective children can be consistently performed and preserves (63). Performing this for each pair of minimal θ-equivalent paths of $\mathcal{V}$ yields the desired trellis. ◇

4.3.2 Operations on observation criteria and trellises

We now introduce important operations on observation criteria:

Definition 8

1. *Let $\theta : L \cup \{\text{-}\} \mapsto \mathcal{L}_\theta$ be an observation criterion, and $L' \subseteq L$. Observation criterion θ is called* L'-consistent *if*

$$\theta(w_1) = \theta(w_2) \quad \Rightarrow \quad \theta\big(\mathbf{proj}_{L' \cup \{\text{-}\}}(w_1)\big) = \theta\big(\mathbf{proj}_{L' \cup \{\text{-}\}}(w_2)\big). \tag{64}$$

   For $L = L' \cup L''$, say that θ is (L', L'')-distributable *if it is both L'- and L''-consistent.*

2. *Two observation criteria $\theta' : L' \cup \{\text{-}\} \mapsto \mathcal{L}_{\theta'}$ and $\theta'' : L'' \cup \{\text{-}\} \mapsto \mathcal{L}_{\theta''}$ are called* compatible *if the two restrictions, of θ' and θ'' to $(L' \cap L'') \cup \{\text{-}\}$, are equal. In this case, we define their* join:

$$(\theta' \sqcup \theta'')(\ell) \quad = \quad \text{if } \ell \in L' \text{ then } \theta'(\ell) \text{ else } \theta''(\ell)$$

   The compatibility of θ' and θ'' ensures that $\theta' \sqcup \theta'' = \theta'' \sqcup \theta'$. In general, $\theta' \sqcup \theta''$ is not (L', L'')-distributable.

3. *For two observation criteria $\theta' : L' \cup \{\text{-}\} \mapsto \mathcal{L}_{\theta'}$ and $\theta'' : L'' \cup \{\text{-}\} \mapsto \mathcal{L}_{\theta''}$ define their* product *as being the following partial function*

$$\theta' \times \theta'' : (L' \cup \{\text{-}\}) \cup (L'' \cup \{\text{-}\}) \mapsto \mathcal{L}_{\theta'} \times \mathcal{L}_{\theta''} \quad : \quad (\theta' \times \theta'')(\ell) = (\theta'(\ell), \theta''(\ell))$$

   $\theta' \times \theta''$ is always (L', L'')-distributable.

A counterexample of $\theta' \sqcup \theta''$ not being L'-consistent is given by $\theta' : L' \cup \{\text{-}\} \mapsto \{1\}^*$ and $\theta'' : L'' \cup \{\text{-}\} \mapsto \{1\}^*$, both counting non silent transitions. Then $\theta' \sqcup \theta'' : L' \cup L'' \cup \{\text{-}\} \mapsto \{1\}^*$ also counts non silent transitions and it is neither L'- nor L''-consistent — take for example $w_1 = \ell_1 \in L' \setminus L''$ and $w_2 = \ell_2 \in L'' \setminus L'$.

The following result indicates how Condition (63) is preserved by the above operations on observation criteria:

Lemma 6 *The following properties hold regarding Condition (63):*

1. *For $\mathcal{A}$ an automaton, its execution tree $\mathcal{U}_\mathcal{A}$ satisfies Condition (63) for any observation criterion θ.*

2. *The set of execution trees satisfying Condition (63) with respect to a given θ is closed under intersection.*

3. *Let θ be an observation criterion, let $\mathcal{V}$ be an (S, L)-execution tree satisfying Condition (63), and let $L' \subseteq L$ be such that θ is L'-consistent. Then the projection $\mathbf{proj}_{L,L';\pi}(\mathcal{V})$ satisfies Condition (63) with respect to θ'.*

4. *If $\mathcal{V}'$ and $\mathcal{V}''$ satisfy Condition (63) with respect to θ' and θ'' and these two observation criteria are compatible, then $\mathcal{V}' \times_U \mathcal{V}''$ satisfies Condition (63) with respect to both $\theta' \times \theta''$ and $\theta' \sqcup \theta''$ (provided that the latter is distributable).*

Proof: We prove the successive statements.

1. Obvious, by definition of $\mathcal{U}_{\mathcal{A}}$.

2. Obvious.

3. Condition (64) ensures that, if two paths σ and σ' of $\mathcal{V}$ are θ-equivalent, then their images by the projection $\mathbf{proj}_{L,L';\pi}$ are also θ'-equivalent. Since equality of child trees is preserved by projection, this statement is proved.

4. Since θ' and θ'' are compatible and $\theta =_{\text{def}} \theta' \sqcup \theta''$ is distributable, then θ is both L'- and L''-consistent. Let σ_1 and σ_2 be two θ-equivalent paths of $\mathcal{V}' \times_U \mathcal{V}''$, and let w_1 and w_2 be the words over labels they define. By (62) and with the corresponding notations, this means that:

$$s_{1,\text{init}} = s_{2,\text{init}} \ , \ s_{1,\text{end}} = s_{2,\text{end}} \ , \text{ and } \theta(w_1) = \theta(w_2). \tag{65}$$

The same holds if we take instead $\theta =_{\text{def}} \theta' \times \theta''$, where the two observation criteria θ' and θ'' are arbitrary. By definition of the product of execution trees, and since θ is distributable, (65) is equivalent to:

$$\begin{array}{c} s'_{1,\text{init}} = s'_{2,\text{init}} \ , \ s'_{1,\text{end}} = s'_{2,\text{end}} \quad , \quad s''_{1,\text{init}} = s''_{2,\text{init}} \ , \ s''_{1,\text{end}} = s''_{2,\text{end}} \\ \theta'(w'_1) = \theta'(w'_2) \quad , \quad \theta''(w''_1) = \theta''(w''_2) \end{array}$$

Therefore, using notation (59), $\mathbf{proj}_{\mathcal{V}'}(\sigma_1)$ and $\mathbf{proj}_{\mathcal{V}'}(\sigma_2)$ are θ'-equivalent, and $\mathbf{proj}_{\mathcal{V}''}(\sigma_1)$ and $\mathbf{proj}_{\mathcal{V}''}(\sigma_2)$ are θ''-equivalent. Thus child trees of $\mathbf{proj}_{\mathcal{V}'}(\sigma_1)$ and $\mathbf{proj}_{\mathcal{V}'}(\sigma_2)$ are isomorphic, and so are child trees of $\mathbf{proj}_{\mathcal{V}''}(\sigma_1)$ and $\mathbf{proj}_{\mathcal{V}''}(\sigma_2)$. Hence, child trees of σ_1 and of σ_2 are isomorphic too. ◇

Using Lemmas 5 and 6, we are now ready to introduce the basic operations on trellises, by building on the corresponding operations, for execution trees:

- For $\mathcal{T}$ and $\mathcal{T}'$ two (S, L, θ)-trellises, their *intersection* is defined by

$$\mathcal{T} \cap \mathcal{T}' \quad =_{\text{def}} \quad \Psi_\theta^{-1}(\Psi_\theta(\mathcal{T}) \cap \Psi_\theta(\mathcal{T}'))$$

- Let $\mathcal{T}$ be an (S, L, θ)-trellis, and let $L' \subseteq L$ be such that θ is L'-consistent, and $\pi : S \mapsto S'$ a total surjection from S onto some alphabet S'. The *projection* of $\mathcal{T}$ onto L' along π is defined by:

$$\mathbf{proj}_{L,L';\pi}(\mathcal{T}) \quad =_{\text{def}} \quad \Psi_{\theta'}^{-1}\left(\mathbf{proj}_{L,L';\pi}(\Psi_\theta(\mathcal{T}))\right), \tag{66}$$

where θ' is the restriction of θ to L'. We shall write simply $\mathbf{proj}_{L';\pi}(\mathcal{T})$ instead of $\mathbf{proj}_{L,L';\pi}(\mathcal{T})$ when no confusion can result. By statement 3 of Lemma 6, this definition is legitimate.

- Let $\mathcal{T}$ be an (S, L, θ)-trellis, and $\mathcal{T}'$ be an (S', L', θ')-trellis. Define the following two kinds of *product*:

$$\text{for } \theta \sqcup \theta' \text{ distributable: } \mathcal{T} \times_{T,\sqcup} \mathcal{T}' \quad =_{\text{def}} \quad \Psi^{-1}_{\theta\sqcup\theta'}\big(\Psi_\theta(\mathcal{T}) \times_U \Psi_{\theta'}(\mathcal{T}')\big) \tag{67}$$

$$\text{for any two } \theta \text{ and } \theta' \text{: } \mathcal{T} \times_{T,\times} \mathcal{T}' \quad =_{\text{def}} \quad \Psi^{-1}_{\theta\times\theta'}\big(\Psi_\theta(\mathcal{T}) \times_U \Psi_{\theta'}(\mathcal{T}')\big) \tag{68}$$

  By statement 4 of Lemma 6, this definition is legitimate. Furthermore, let $\sigma_i, i = 1, 2$ and $\sigma'_i, i = 1, 2$ be two pairs of equivalent paths of $\Psi_\theta(\mathcal{T})$ and $\Psi_{\theta'}(\mathcal{T}')$, respectively, such that σ_1 and σ'_1, on the one hand, and σ_2 and σ'_2, on the other hand, synchronize to yield two paths of $\Psi_\theta(\mathcal{T}) \times_U \Psi_{\theta'}(\mathcal{T}')$. Then these two paths are also equivalent, for the two observation criteria $\theta' \sqcup \theta''$ and $\theta' \times \theta''$.

The following result aims at satisfying Requirement 1:

Theorem 7 *The above operations of intersection, projection, and product, can be computed directly on trellises.*

Proof: See Algorithm 7 of Appendix A.3. ◇

4.3.3 Discussion: interleaving versus partial orders

In this section we compare the two kinds of products, in terms of efficiency of the resulting data structure.

Consider first the case in which each local system $\mathcal{A}_i$ is equipped with observation criterion

$$\theta_i \quad : \quad L_i \mapsto \{1\}^*. \tag{69}$$

Observation criterion θ_i counts the transitions (for simplicity, we assume that all labels of L_i are observed). Note that these observation criteria are pairwise compatible, so that we can consider both their join $\Theta_\sqcup = \bigsqcup_{i\in I} \theta_i$ and their product $\Theta_\times = \bigtimes_{i\in I} \theta_i$.

While $\Theta_\times$ is distributable, $\Theta_\sqcup$ is not. The following problem occurs when using $\Theta_\sqcup$, see Fig. 7 for the case whare I has cardinal two. This figure shows two automata $\mathcal{A}$ and $\mathcal{A}'$. Note that $\mathcal{A}'$ itself is already a θ'-trellis. The last diagram shown is obtained by performing projection as explained. It does not yield a valid trellis, however, since the two paths $s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_2$ and $s_0 \xrightarrow{c} s_2$ shoud not be confluent because they have different lengths. The reason is that $\Theta_\sqcup$ is not distributable. This problem disappears when (correctly) using $\Theta_\times$, see Fig. 8.

Let us modify the observation criteria as follows. Take

$$\begin{array}{rrrrrr} \theta(\ell) = \ell & \text{if} & \ell = e, & \text{else} & \theta(\ell) = \text{-} \\ \theta'(\ell') = \ell' & \text{if} & \ell' = d, & \text{else} & \theta'(\ell') = \text{-} \end{array} \tag{70}$$

In words, θ counts the number of e's and mark them with symbol "e", and θ' counts the number of d's and mark them with symbol "d". This amounts to considering that only

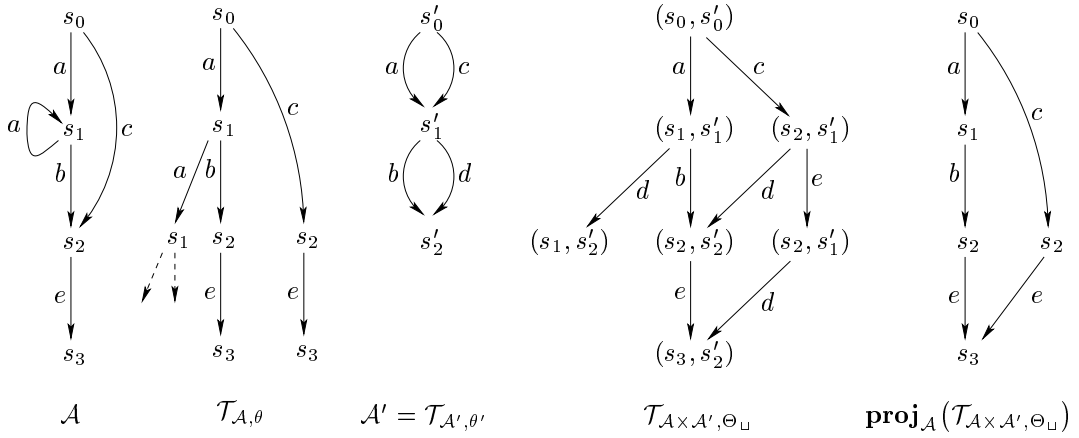

Figure 7: Illegal use of $\Theta_\sqcup$ when local observation criteria are given by (69).

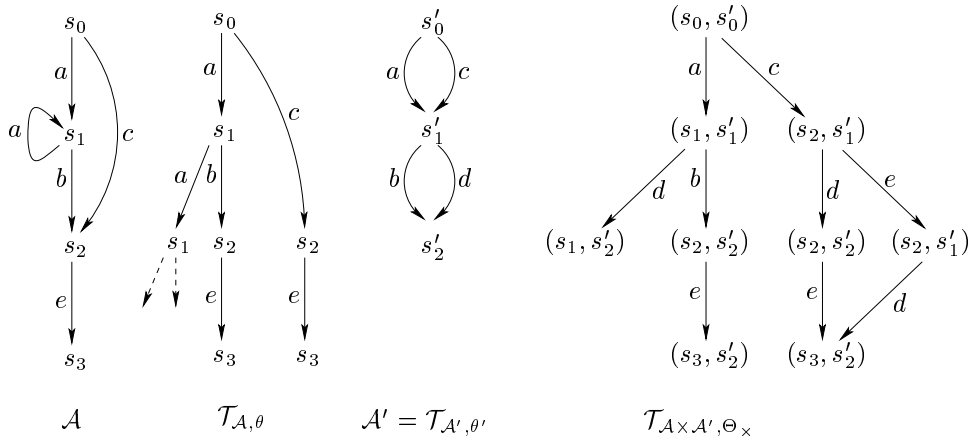

Figure 8: Always legal use of $\Theta_\times$.

observed events are counted and $L_o = \{e\}, L'_o = \{d\}$. Since $L_o \cap L'_o = \emptyset$, it follows that θ and θ' are compatible, hence $\Theta_\sqcup$ is well defined. Since θ and θ' map symbols to disjoint sets, $\Theta_\sqcup$ is now distributable and can thus be legally used. The two observation criteria $\Theta_\sqcup$ and $\Theta_\times$ are compared in Figure 9. The latter is more efficient: $\Theta_\sqcup$ distinguishes paths that differ by interleaving, whereas $\Theta_\times$ does not. This is the reason for the merge on the second diagram.

The conclusion is that one should always use product $\times_{T,\times}$, never $\times_{T,\sqcup}$, for the following two reasons: the former is legal for any tuple of local observation criteria, and even when

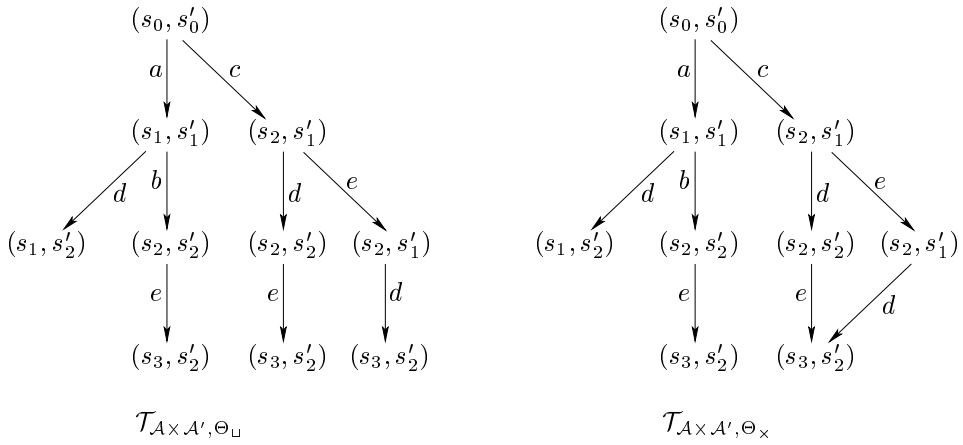

Figure 9: Comparing legal products using $\Theta_\sqcup$ and $\Theta_\times$, for observation criteria (70).

the latter is legal, it is less efficient. Vector observation criteria are preferred to interleaving ones.

Relation with Fidge-Mattern vector clocks for distributed systems. *Vector clocks* have been introduced for the analysis of distributed systems and algorithms in the 80's by Mattern [36] and Fidge [22]. Using vector clocks amounts to regarding executions of the overall distributed system as tuples of synchronized local executions. Product $\times_{T,\times}$ amounts to using vector clocks, *which is nothing but taking a partial order view of distributed executions, where local executions are still considered sequential.*

4.3.4 Trellis based monitors

We want to extend our algebraic formulation of monitors to trellises. Considering the analysis of the previous section, we shall only use product $\times_{T,\times}$, which we denote simply by $\times_T$.

We first discuss global monitoring. Let $\mathcal{A} = (S, L, \to, S_0), L = L_o \cup L_u$ be an automaton, and θ an observation criterion for it. Let Ω be a set of observations for $\mathcal{A}$, represented as an $(\mathbb{N}, L_o, \theta_o)$-trellis. The trellis based monitor for $\mathcal{A}$ is defined as the map

$$\Omega \quad \longmapsto \quad \mathcal{M} \;=_{\text{def}}\; \mathbf{proj}_\pi\big(\mathcal{T}_{\mathcal{A},\theta} \times_T \Omega\big), \tag{71}$$

where projection π removes the state label arising from the observations.

Next, thanks to Lemmas 5 and 6, the apparatus composed of factorization Theorem 1, partial order $\preceq$, and Lemmas 1–3 carries over to trellises. Having this at hand, we can next consider *modular monitoring* with trellises.

Let $\mathcal{A} = \times_{i\in I}\mathcal{A}_i$ be a product of automata and let $\theta_i, i \in I$, be a family of observation criteria and set $\Theta = \times_{i\in I}\theta_i$. For each $i \in I$, let $\theta_{o,i}$ be the restriction of θ_i to $L_{o,i}$ and set $\Theta_o = \times_{i\in I}\theta_{o,i}$. For each i, let ω_i be an observation for $\mathcal{A}_i$. It is a chain, and thus we can see it as a trellis with observation criterion $\theta_{o,i}$. The global observation is thus represented by the $(\mathbb{N}^I, L_o, \Theta_o)$-trellis

$$\Omega \quad = \quad \times^T_{i\in I}\,\omega_i.$$

Then, having a factorization theorem for trellises, monitoring can again be defined as the map:

$$(\omega_i)_{i\in I} \quad \longmapsto \quad \mathcal{M}_{\mathrm{mod}} =_{\mathrm{def}} \left(\mathbf{proj}_{L_i;\pi_i}\left(\mathcal{T}_{\mathcal{A},\Theta} \times_T \Omega\right)\right)_{i\in I}, \tag{72}$$

where $\pi_i : S\times\mathbb{N}^I \mapsto S_i$ is the projection over the ith local state of the system for monitoring. Examples of distributable observation criteria for modular monitoring are cases (ii) and (iii) of Section 4.3.3.

5 From trellises to partial order models

In the preceding section, we have seen that runs of distributed systems should be seen as partial orders, obtained by synchronizing the sequential runs of components. Now, if the components of the distributed system interact asynchronously, then internal concurrency also must exist within each component. Hence, the runs of a component should themselves be seen as partial orders. Thus it makes sense to construct a variant of unfoldings or trellises, where runs appear as partial orders. This is illustrated in Fig. 10. Advantages and difficulties are discussed next.

Advantages:

- Partial order unfoldings are better than interleaving ones in that they remove diamonds within the component or system considered. This causes reduction in size.
- Furthermore, when long but finite runs are considered for the monitoring problem, it may be that partial order unfoldings perform nearly as well as interleaving based trellises; this is, *e.g.,* the case when most merge in the considered trellis originate from diamonds in the interleaving semantics.
- Partial order trellises are better than interleaving ones in that they remove diamonds within the component or system considered. This causes reduction in size.
- Partial order unfoldings and trellises can be equipped with notions of product and intersection.

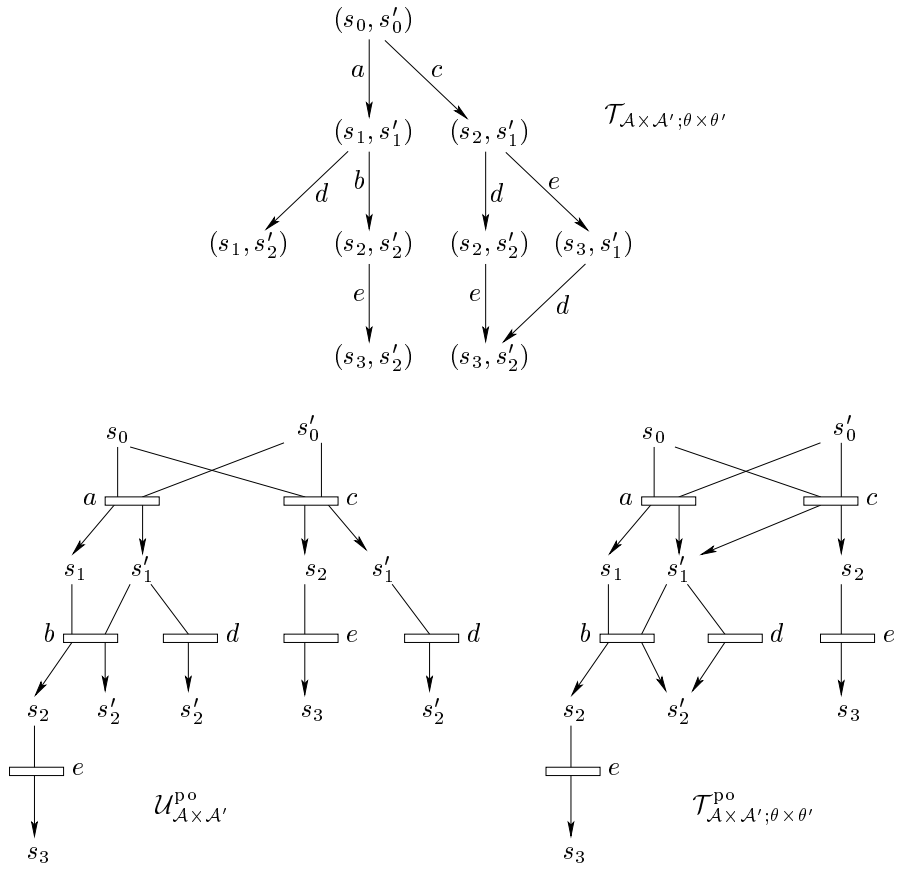

Figure 10: Showing the partial order unfolding $\mathcal{U}^{\mathrm{po}}_{\mathcal{A}\times\mathcal{A}'}$ and trellis $\mathcal{T}^{\mathrm{po}}_{\mathcal{A}\times\mathcal{A}';\theta\times\theta'}$; for comparison, we have left the sequential trellis $\mathcal{T}_{\mathcal{A}\times\mathcal{A}';\theta\times\theta'}$. Note that the diamond has disappeared in both cases.

Difficulty: the projection of a partial order unfolding or trellis can sometimes *not* be represented as another partial order unfoldings or trellis, see Fig. 11. This figure shows the problem with partial order unfoldings, but the same difficulty holds with partial order trellises.

Solutions when using partial order unfoldings. When using partial order unfoldings, the difficulty can be circumvent by one of the following means:

- 1st *method:* enhance occurrence nets with possible additional causalities and conflicts, not resulting from the graph structure of the net. This is the approach taken in [15, 16].

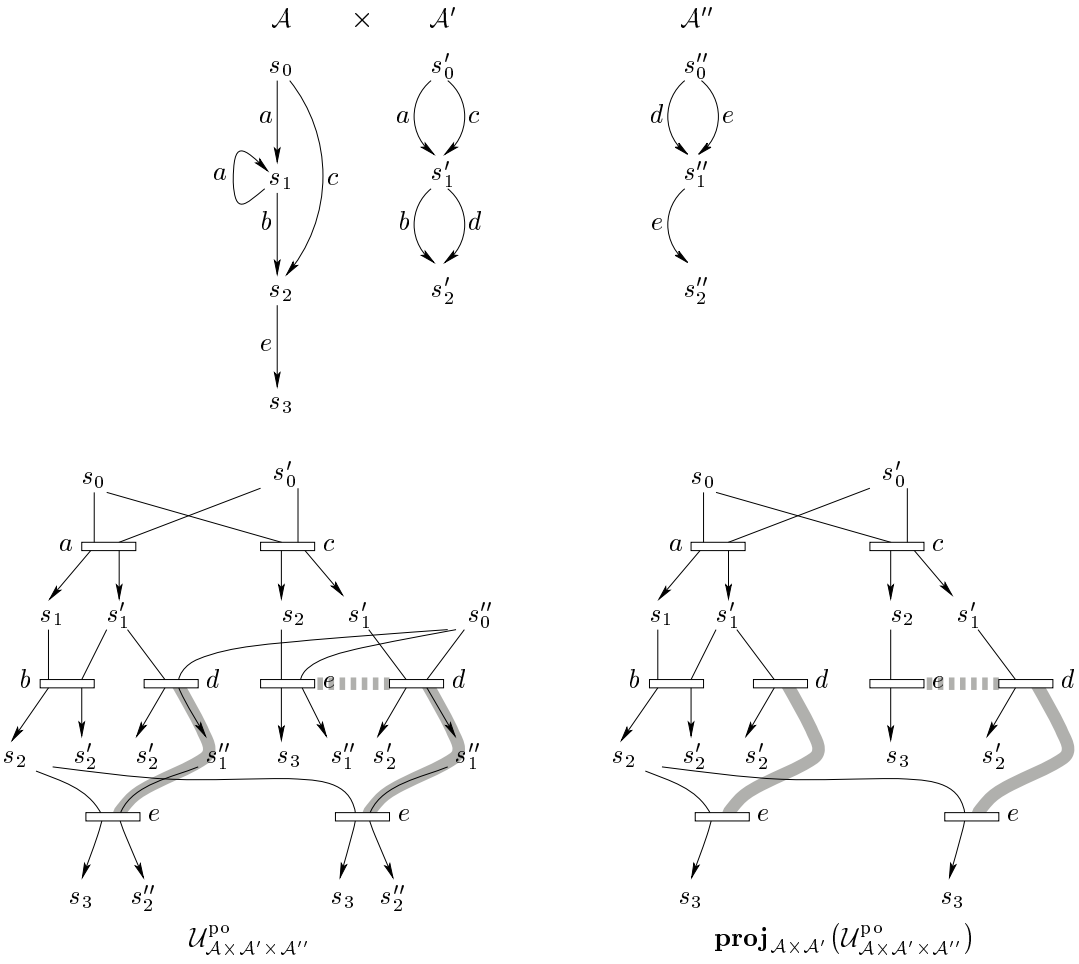

Figure 11: The figure shows a distributed system with two components, written as $(\mathcal{A}\times\mathcal{A}')\times\mathcal{A}''$. This means that the first component is already a distributed system and therefore has internal concurrency. We show on the right the partial order unfolding of this distributed system. Some **conflicts** are depicted in in thick gray dashed lines and some **causalities** are depicted in thick gray solid lines. Projecting on the first component should yield the last diagram, having the conflicts and causalities in it. Unfortunately, these cannot be captured by occurrence net features, with the available nodes. An enriched structure is needed.

- 2nd *method:* abandon occurrence nets and use *event structures* instead. Event structures are sets of events equipped directly with a causality relation and a conflict rela-

tion, with no use of condition nodes to graphically encode conflict. This is the approach taken in [18].

- 3rd *method:* keep occurrence nets as such, but avoid the enhancement used in the 1st method by exchanging messages in the form of so-called *interleaving structures*, see [4].

With these modifications, the preceding techniques for distributed monitoring with partial order unfoldings apply. The development of similar techniques for partial order trellises is under progress.

6 Related work on distributed diagnosis

Distributed diagnosis has been investigated within the so-called Discrete Event Systems community. Most of the work performed consists in extensions and adaptations of the decentralized diagnosis framework originally introduced by Debouk, Lafortune, and Teneketzis [11]. This early work introduces the idea of distributed observers, although modularity of computations is not fully developed, since the underlying system is handled as a whole. Boel and van Schuppen [7] have examined an asymmetric version of this setting, where one observer helps the second, and at the same time minimizes its communication cost. The effect of delays in communication channels is explicitly studied and handled [12, 41], and issues of decidability of distributed observability (or diagnosability) were analyzed by Tripakis [49]. By contrast with the above contributions, the work of Su and Wonham [46, 47, 48], that we extensively discussed, fully investigates modularity issues and is probably the closest to the present paper. It introduces the notion of supremal local support of a language system. Moreover, this object is computed by a message passing algorithm, as in our case. The on-line version is not investigated however. All these studies use the classical automata/languages/product paradigm, which makes considering distributed systems more difficult. The recent line of search by Lafortune on Petri net diagnosis introduces an algebra for distributed systems that is closer to ours [23, 24].

Diagnosis has also been investigated in the AI community, see in particular [28]–[31] and [39]. Most interesting is the book [29]. In this work, the same problem of monitoring is considered as in our paper. The solutions are stored and manipulated in the form of labeled Directed Acyclic Graphs resembling our unfoldings. One step further is performed compared to unfoldings: when an unobserved cycle of the automaton exists, then it is kept as such in this "partial unfolding", very much like what we did in Figure 6 for trellises. Compared to the present work, Lamperti-Zanella's one does not attempt to formalize the data structures they use. As a consequence, distributed algorithms become cumbersome and their correctness is difficult to verify. This fact is indeed a strong argument in favor of our more algebraic approach to deal with data structures.

The message passing algorithms we have developed in Sections 3.4 and 3.5 relate to so-called *belief propagation* algorithms in the area of Bayesian Networks, a community bridging AI and statistics [32, 38]. These ideas are nevertheless present in many communities, under different names (signal and image processing, digital communications, coding theory, etc.).

The algebraic techniques we used to manipulate data structures originate from a totally different community. Foundations are found in the seminal work [37] on event structure semantics of Petri nets. Unfolding theory and event structures have been subsequently developed by Winskel, e.g., in [52, 53]. The interest of the partial order nature of unfoldings has been first recognized by McMillan [34, 35] in the context of model checking. Systematic investigation of factorization properties of data structures, and their use in distributed algorithms were then explored in our group [15, 16, 19].

7 Extensions and further research issues

In this section we review some further problems arising from applications and we draw corresponding research directions.

7.1 Building models for large systems: self-modeling

As explained in Appendix B, realistic applications such as fault management in telecommunication networks and services require models of complexity and size far beyond what can be constructed by hand. Thus, any model based algorithm would fail addressing such type of application unless proper means are found to construct the model.

In some contexts including the one reported in Appendix B, an automatic construction is possible. One approach developed in [1] is called *self-modeling*. Its principle is illlustrated

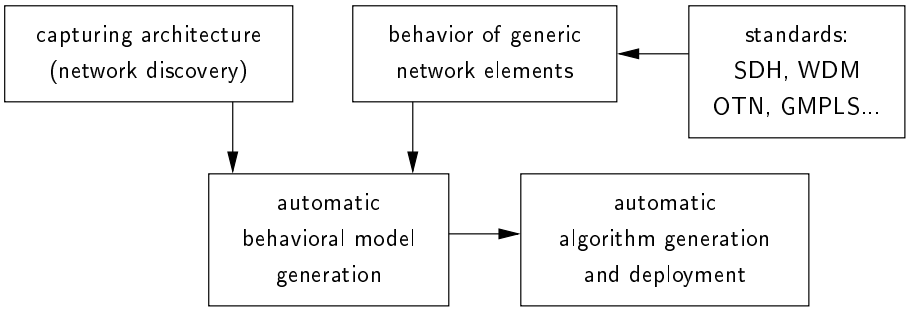

Figure 12: Self-modeling.

in Fig. 12. To construct models, the following prior information is assumed available:

(a) *A finite set of prototype components is available, and all systems considered are obtained by composing instances of these prototype components.*

In our application context, these prototype components are specified by the different network standards used (as listed in the left most box of Fig. 12), in the form of an inheritance tree of *Managed Classes*, described in the so-called information model of each technology. In this context, the number of classes for consideration is typically

small (a dozen or so). In contrast the number of instantiated components in the systems may be huge (from hundreds to thousands).

(b) *For each prototype component, a behavioral model is available in one of the forms we discussed in this paper.*

This is the manual part of the modeling. It was done, *e.g.,* by Alcatel, for the case of all standards shown in the left most box of Fig. 12, by browsing component behaviour descriptions in the norms, and parsing typical failure scenarios [1].

(c) *System architecture can be automatically discovered.*

By "system architecture" we mean the structure of the system (list of instances and their topology and interconnections). This assumes that so-called reflexive architectures are used, *i.e.,* architectures carrying a structural model of themselves. This is for example the case in our context, where this task is referred to as *network discovery.*

Having (a), (b), and (c) allows to construct automatically the system model $(\mathcal{A}_i)_{i\in I}$ and even generate and deploy the monitoring algorithm automatically [1].

7.2 Probabilistic true concurrency models

In real-life applications, monitoring and diagnosis generally yield ambiguous results. For example, in real-life systems, multiple faults must be considered; as a result, it is often possible to explain the same observations by either one single fault or two independent faults. This motivates considering probabilistic models and developing maximum likelihood algorithms.

In doing this, we would obviously like that noninteracting subsystems are probabilistically independent. None of the classical probabilistic DES models (Markov chains, Hidden Markov Models, Stochastic Petri nets, stochastic automata) has this property. Samy Abbes [2, 3] has developed the fundamentals of true concurrency probabilistic models.

7.3 Timed true concurrency models

In performing monitoring or diagnosis, physical time (even imprecise) can be used to filter out some configurations. *Timed systems* models are needed for this. Candidates are timed automata and concurrent or partial order versions of them [9].

7.4 Dynamically changing systems — Objective 4

So far we mentioned this objective but did not address it in this paper. In fact, addressing it is the very motivation for considering run-based on-line algorithms in which no diagnoser is statically pre-computed. Models of dynamically changing DES are not classical. A variety of them have been proposed in the context of distributed systems. *Petri net systems* [14] are systems of equations relating Petri nets; these models allow for dynamic instantiation of predefined nets. Variants of such models exist in the Petri net litterature. *Graph Grammars* [44]

are more powerful as they use a uniform framework to represent both the movement of tokens in a net and the creation/deletion of transitions or subnets in a dynamic net. Graph Grammars have been used by Haar et al. [25] for diagnosis under dynamic reconfiguration. This subject is still in its infancy.

7.5 Incomplete models

For large, real-life systems, having an exact model (*i.e.,* accepting all observed runs while being at the same time non trivial) can hardly be expected. The kind of algorithm the DES community develops gets stuck when no explanation is found for an observation. In contrast, pattern matching techniques such as *chronicle recognition* [13] developed in the AI community are less precise than the DES model based techniques but do not suffer from this drawback. Leveraging the advantages of DES model based techniques to accept incomplete models is a challenge that must be addressed.

8 Conclusion

We have discussed diagnosis of large networked systems. Our research agenda and requirements setting were motivated by the context of our ongoing cooperation with Alcatel, as briefly reported in the appendix. The focus of this paper was on on-line distributed diagnosis, where diagnosis is reported in the form of a set of hidden state histories explaining the recorded alarm sequences. In this context, efficiency of data structures to represent sets of histories is a key issue.

We have tried to deviate least possible from the classical setting, where distributed systems are modeled through the parallel composition of automata or languages. Our conclusion is that, to a certain extend, adopting a partial order viewpoint cannot be avoided. To the least, distributed executions must be seen as a partial order of interacting concurrent sequences of events. Of course, adopting a truly concurrent setting in which executions are systematically represented as partial orders is also possible.

This heterodox viewpoint raises a number of nonstandard research issues, some of which were listed in the previous section. While our group has started addressing some of these, much room remains for further research in this exciting area.

Another important remark we like to state is the usefulness of categorical techniques in analysing the issues we discussed in this paper. Note that we have considered a large variety of data structures to represent sets of runs. For each of them, we have considered the wished set of basic operators. Getting the desired factorization properties can become a real nightmare if only pedestrian techniques are used — see, *e.g.,* [18] for such a situation. In contrast, taking a categorical perspective [33] significantly helps structuring the research problems and focusing on the right properties for checking. It also prevents the researcher from redoing variants of her proofs. See for instance [4, 19, 20].

A Appendix: effective algorithms

A.1 Product of chain processes

Let Σ_1, Σ_2 be chain processes, and Σ denote their product $\Sigma_1 \times_C \Sigma_2$. The algorithm below recursively builds the prefix closure $[\Sigma]$ of Σ, which can then be "filtered out" to remove spurious runs of $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$ not belonging to $\Sigma_1 \times_C \Sigma_2$.

Notations. Let σ_i be a chain in Σ_i, representing a run of $\mathcal{A}_i$. We denote by $[\sigma_i\rangle s_i$ the fact that σ_i leads to state s_i of $\mathcal{A}_i$. And $\sigma_i' = \sigma_i \bullet l_i \bullet s_i'$ denotes the extension of σ_i with the extra transition $s_i \xrightarrow{l_i} s_i'$ of $\mathcal{A}_i$. We take $L = L_1 \cup L_2$, $S = S_1 \times S_2$ and $\pi_i : S \to S_i$ the canonical projection.

Algorithm 5 (product of chain processes Σ_1 and Σ_2)

- *Initialization.* $\Sigma = \{\epsilon\}$ *where* ϵ *denotes the empty chain.*
- *Recursion. Apply the following extension rule until stability of* Σ*: let* $\sigma \in \Sigma$*, with* $\sigma_i = \mathbf{proj}_{L,L_i;\pi_i}(\sigma)$*,* $[\sigma_i\rangle s_i$*, and let* $l \in L$*:*
  - *if* $l \in L_1 \cap L_2$ *and* $\exists \sigma_i' = \sigma_i \bullet l \bullet s_i' \in [\Sigma_i]$*,* $i = 1, 2$*,*
    then let $\sigma' = \sigma \bullet l \bullet (s_1', s_2')$ *and* $\Sigma := \Sigma \cup \{\sigma'\}$*,*
  - *if* $l \in L_1 \setminus L_2$ *and* $\exists \sigma_1' = \sigma_1 \bullet l \bullet s_1' \in [\Sigma_1]$*,*
    then let $\sigma' = \sigma \bullet l \bullet (s_1', s_2)$ *and* $\Sigma := \Sigma \cup \{\sigma'\}$*,*
  - *symmetrically for* $l \in L_2 \setminus L_1$*.*

The filtering of Σ can be performed afterwards, by simply removing runs σ such that their projections σ_i are not in Σ_i. This operation could also be incorporated to algorithm 5, which we didn't do for clarity: given σ such the $\sigma_i \notin \Sigma_i$ for $i = 1$ or $i = 2$, remove σ from Σ as soon as all possible extensions by $l \in L$ have been tried.

A.2 Product of execution trees

We base the construction procedure on definition (58) that derives $\times_U$ from $\times_C$ by

$$\mathcal{V}_1 \times_U \mathcal{V}_2 \quad = \quad \Phi^{-1}(\Phi(\mathcal{V}_1) \times_C \Phi(\mathcal{V}_2))$$

So the essential modification in algorithm 5 amounts to incorporating the "refolding" Φ^{-1} of Σ into an execution tree.

Notations. We denote by n_i a generic node of the tree $\mathbf{T}_i$ of $\mathcal{V}_i = (\mathbf{T}_i, \lambda_i, f_i)$. Node n_i identifies the unique run of the prefix closure $[\mathcal{V}_i]$ of $\mathcal{V}_i$ ending at node n_i, we denote this run by $\downarrow n_i$ ($\downarrow n_i$ is the causal past of node n_i in $\mathcal{V}_i$). For n a node of $\mathcal{V} = \mathcal{V}_1 \times_U \mathcal{V}_2$, corresponding to run $\sigma = \downarrow n$ of $[\mathcal{V}]$, we denote by $n_i =_{\text{def}} \chi_i(n)$ the node of $\mathcal{V}_i$ that corresponds to $\sigma_i = \mathbf{proj}_{L,L_i;\pi_i}(\sigma)$.

Algorithm 6 (product of execution trees $\mathcal{V}_1$ and $\mathcal{V}_2$)

- *Initialization:* $\mathcal{V} = (\mathbf{T}, \lambda, f)$ *where*
  - $\mathbf{T} = \{r\}$, *a single rootnode, no paths, with*
  - $\chi_i(r) = r_i$, *the rootnode of* $\mathbf{T}_i$, $i = 1, 2$
  - $\lambda(r) = (\lambda_1(r_1), \lambda_2(r_2))$,
  - $f(r) = f_1(r_1)f_2(r_2)$, i.e. r *is a stop point iff both* r_i *are stop points.*
- *Recursion: until stability of* $\mathcal{V}$, *apply the following extension rule*
  let n *be a node of* $\mathbf{T}$, $n_i = \chi_i(n)$, $i = 1, 2$, *and*
  let $l \in L$ *such that path* $n \xrightarrow{l} n'$ *does not exist in* $\mathcal{V}$
  - *if* $l \in L_1 \cap L_2$ *and* $\exists n_i \xrightarrow{l} n_i' \in [\mathcal{V}_i]$, $i = 1, 2$,
    then create $n \xrightarrow{l} n'$ *in* $\mathcal{V}$
    with $\chi_i(n') = n_i'$, $i = 1, 2$, $\lambda(n') = (\lambda_1(n_1'), \lambda_2(n_2'))$ *and* $f(n') = f_1(n_1')f_2(n_2')$,
  - *if* $l \in L_1 \setminus L_2$ *and* $\exists n_1 \xrightarrow{l} n_1' \in [\mathcal{V}_1]$,
    then create $n \xrightarrow{l} n'$ *in* $\mathcal{V}$
    with $\chi_1(n') = n_1'$, $\chi_2(n') = n_2$, $\lambda(n') = (\lambda_1(n_1'), \lambda_2(n_2))$ *and* $f(n') = f_1(n_1')f_2(n_2)$,
  - *symmetrically for* $l \in L_2 \setminus L_1$.

In the very same way that Algorithm 5 was building the prefix closure of $\Sigma_1 \times_C \Sigma_2$, this procedure introduces spurious or "dead" paths in $\mathcal{V}$, *i.e.* paths that do not lead to a run of $\mathcal{V}$. The edge $n \xrightarrow{l} n'$ is dead in $\mathcal{V}$ iff the subtree beyond n' (including n') is finite and doesn't contain any stop point. Such paths must be removed after convergence of algorithm 6. They could also be detected and discarded on the fly: when no extension is possible after some node n' that is not a stop point, the edge $n \xrightarrow{l} n'$ (or the node n') is declared dead. Similarly, a node that is not a stop point and leads only to dead nodes is dead itself. This can be easily implemented in algorithm 6 under the form of an extra backtracking rule, which we don't do for clarity. The essential point being that $\mathcal{V}_1 \times_U \mathcal{V}_2$ can be computed directly, without the need to perform the unwrapping Φ on it.

A.3 Product of trellises

As above, we base the construction on definition (67), that derives $\times_{T,\sqcup}$ from $\times_U$ by

$$\mathcal{T}_1 \times_{T,\sqcup} \mathcal{T}_2 \quad = \quad \Psi_\theta^{-1}(\Psi_{\theta_1}(\mathcal{T}_1) \times_U \Psi_{\theta_2}(\mathcal{T}_2))$$

where $\theta = \theta_1 \sqcup \theta_2$ and is distributable. (The approach is identical for the other product $\times_{T,\times}$ and $\theta = \theta_1 \times \theta_2$.) The essential modification with respect to algorithm 6 is thus the introduction of the "refolding" performed by Ψ_θ^{-1} of $\mathcal{T}$ into a trellis.

Notations. Recall that a node n_i in trellis $\mathcal{T}_i$ now represents a set of θ_i-equivalent runs σ_i, for which n_i is the maximal node. Given node n in $\mathcal{T}$, extremity of a run σ, we still denote by $\chi_i(n) = n_i$ the extremal node of $\sigma_i = \mathbf{proj}_{L,L_i;\pi_i}(\sigma)$. By definition of trellises, n_i doesn't depend on which σ ending at n is selected.

Algorithm 7 (product of trellises $\mathcal{T}_1 \times_{_{T,\sqcup}} \mathcal{T}_2$)

- *Initialization:* $\mathcal{T} = (\mathbf{G}, \lambda, f, \theta)$ *where* $(\mathbf{G}, \lambda, f)$ *is as in algorithm 6 up to notations, and* $\theta = \theta_1 \sqcup \theta_2$.
- *Recursion: until stability of* $\mathcal{T}$*, apply the following extension rule*
  - *same as in algorithm 6, but after the creation of a path* $n \xrightarrow{l} n'$ *in* $\mathcal{T}$*, if* $\exists n'' \in \mathcal{T}$ *such that* n' *and* n'' *represent* θ*-equivalent (sets of) runs, then merge* n' *and* n''.

Although this procedure will obviously yield a valid (S, L, θ)-trellis, by construction, we must however justify that the merge is legal in the recursion.

First of all, observe that if n' and n'' are θ-equivalent, then $n'_i = \chi_i(n')$ and $n''_i = \chi_i(n'')$ are θ_i-equivalent, thanks to the assumption that $\theta = \theta_1 \sqcup \theta_2$ is distributable. So one has $n'_i = n''_i$ by definition of an (S_i, T_i, θ_i)-trellis, and χ_i is well defined on the "merged node" (n', n''). This also shows that the labelings $\lambda(n')$ and $\lambda(n'')$ are identical. And in the same way, the stop values $f(n')$ and $f(n'')$ are also identical.

As in algorithm 6, spurious/dead paths may be built in the recursion. They can be discarded at convergence of algorithm 7 or at runtime, in the same manner.

B Appendix, application context: distributed fault management in telecommunications networks

The techniques reported in this paper were developed in the context of a cooperation with the group of Armen Aghasaryan at Alcatel Research and Innovation. A demonstrator has been developed for distributed fault diagnosis and alarm correlation within the ALMAP ALcatel MAnagement Platform.

More recently, an exploratory development has been performed by Armen Aghasaryan and Eric Fabre for the Optical Systems business division of Alcatel. The system considered is shown in Fig. A.1. In this application, diagnosis is still performed centrally, but the system for monitoring is clearly widely distributed. Diagnosis covers both the transmission system (optical fiber, optical components) and the computer equipment itself. Fault propagation was not very complex but self-modeling proved essential in this context. Performance of the algorithms was essential.

A typical use case of distributed monitoring is illustrated if Figs. A.2–4. Fig. A.2 illustrates cross-domain management and impact analysis. The network for monitoring is the optical ring of Paris area with its four supervision centers. When a fault is diagnosed, its

possible impact on the services deployed over it is computed — this is another kind of model based algorithm.

As for the optical ring itself, Fig. A.3 shows the system for monitoring. It is a network of several hundreds of small automata — called *managed objects* — having a handful of states and interacting asynchronously. Due to the object oriented nature of this software system, each managed object possesses its own monitoring system. This monitoring system detects failures to deliver proper service; it receives, from neighboring components, messages indicating failure to deliver service and sends failure messages to neighbors in case of incorrect functioning. This object oriented monitoring system causes a large number of redundant alarms travelling within the management system and subsequently recorded by the supervisor(s). Fig. A.4 shows a typical fault propagation scenario involving both horizontal (across physical devices) and vertical (across management layer hierarchy) propagation.

The problem of recognising causally related alarms is called *alarm correlation.* Fig. A.5 shows how monitoring results are returned to the operator, by proposing candidate correlations between the thousands of alarms recorded, *i.e.,* which alarm causally results from which other alarm. This shows by the way that diagnosis is not necessarily formulated, in real life applications, as that of isolating specific pre-defined faults.

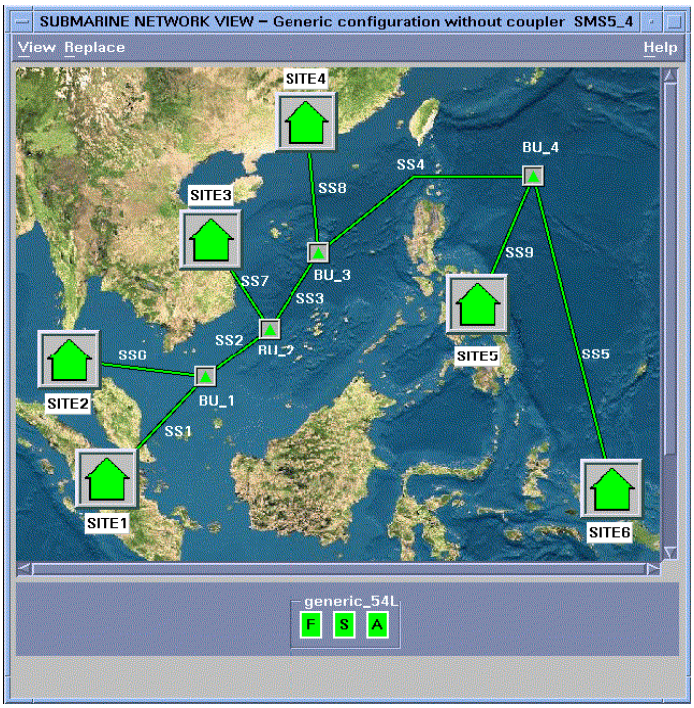

Figure A.1: the submarine optical telecommunication system considered for the trial with Alcatel Optical Systems business division and Alcatel Research and Innovation.

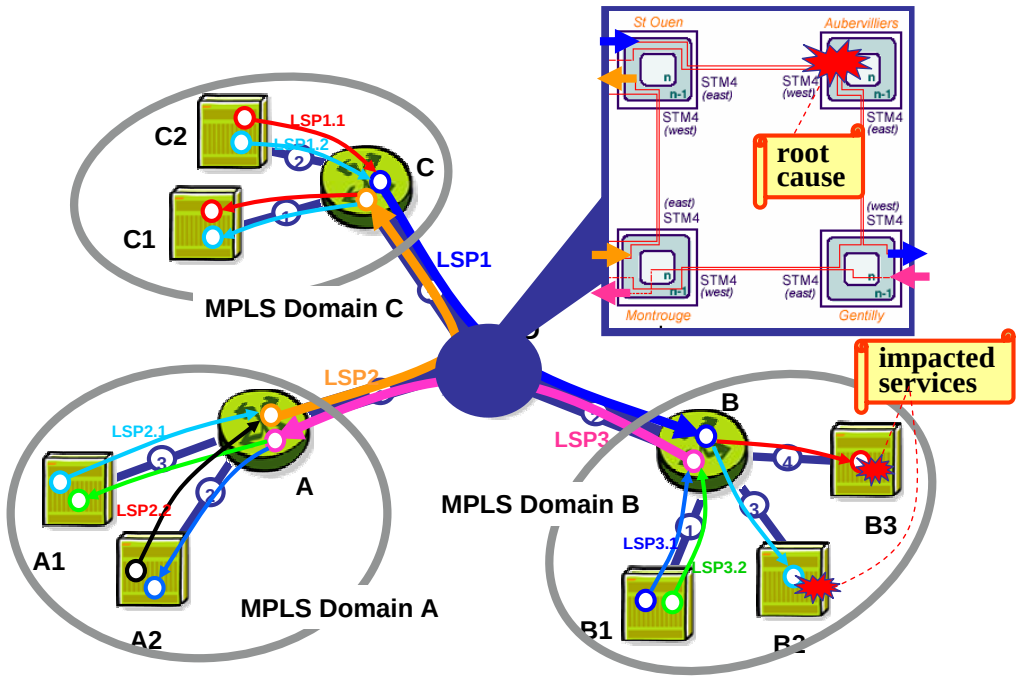

Figure A.2: failure impact analysis.

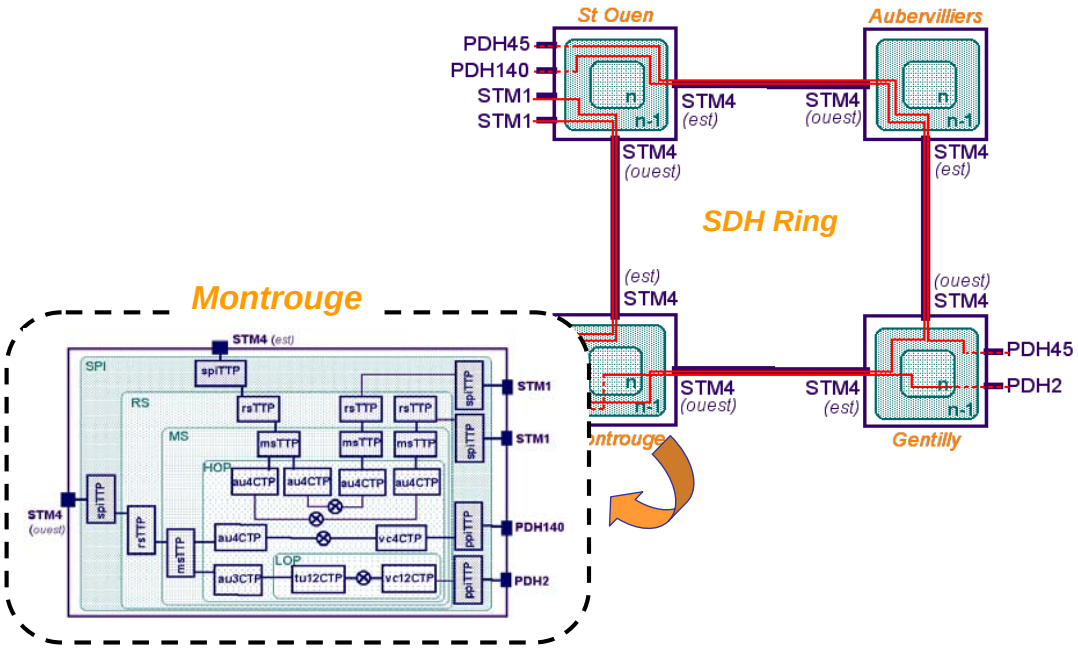

Figure A.3: the SDH/SONET optical ring of the Paris area, with its four nodes. The diagram on the left zooms on the structure of the management software, and shows its Managed Objects

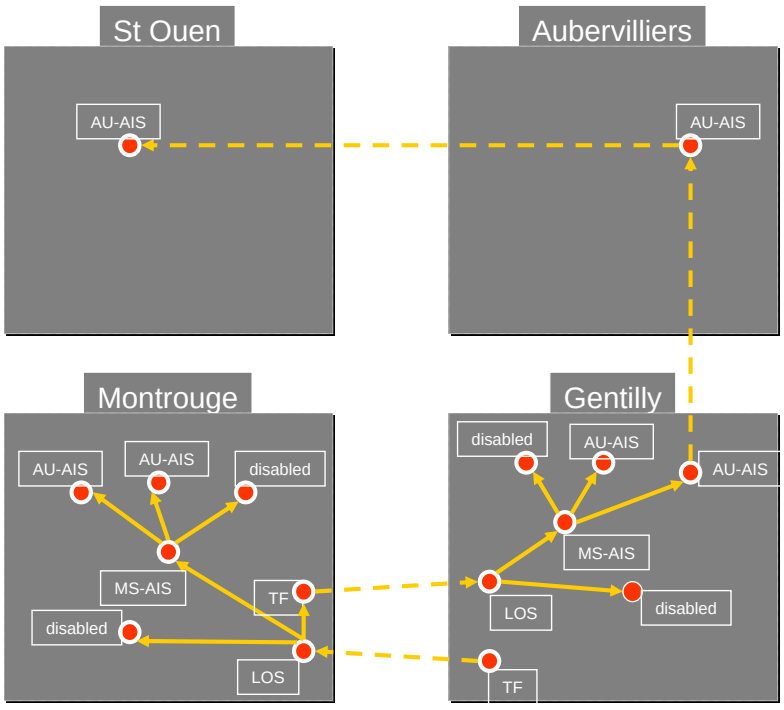

Figure A.4: showing a failure propagation scenario, across management layers (vertically) and network nodes (horizontally).

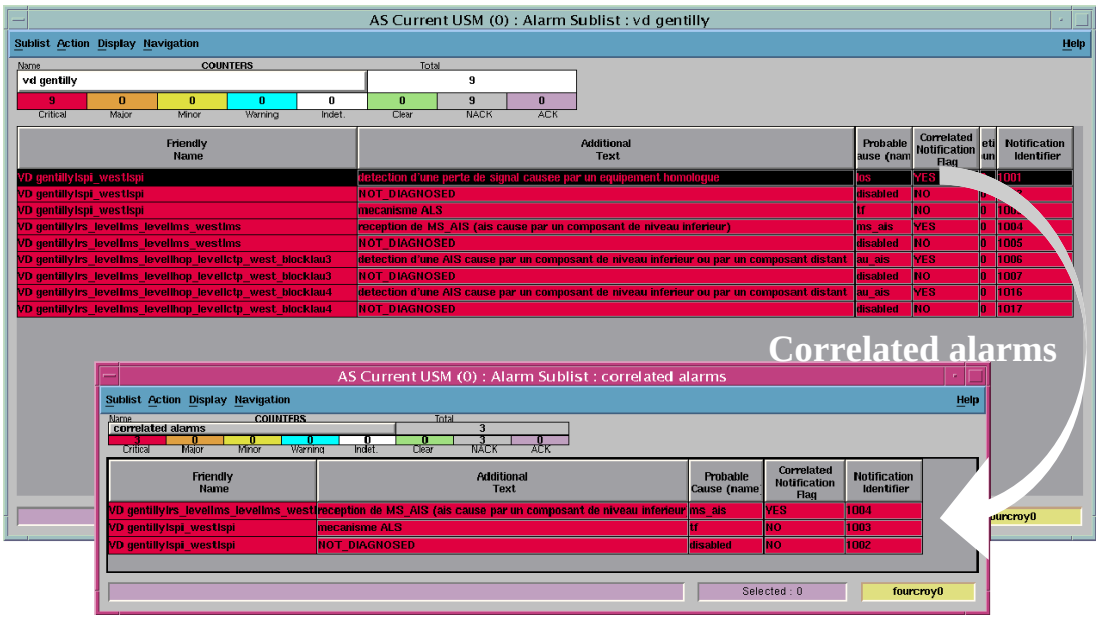

Figure A.5: returning alarm correlation information to the operator.

References

[1] A. Aghasaryan, C. Jard, J. Thomas. UML Specification of a Generic Model for Fault Diagnosis of Telecommunication Networks. In International Communication Conference (ICT), LNCS 3124, Pages 841-847, Fortaleza, Brasil, August 2004.

[2] S. Abbes, A. Benveniste. Branching Cells as Local States for Event Structures and Nets: Probabilistic Applications, in: FoSSaCSV. Sassone (editor), 2005, vol. 3441, pp. 95–109.

[3] S. Abbes and A. Benveniste. True-concurrency Probabilistic Models: Branching cells and Distributed Probabilities for Event Structures. *Information and Computation*, 204 (2), 231-274. Feb 2006.

[4] P. Baldan, S. Haar, and B. König. Distributed Unfolding of Petri Nets. Proc. of FOSSACS 2006, LNCS 3921, pp 126-141, Springer 2006.

[5] P. Baroni, G. Lamperti, P. Pogliano, M. Zanella, Diagnosis of Large Active Systems, Artificial Intell. 110, pp. 135-183, 1999.

[6] A. Benveniste, E. Fabre, S. Haar, C. Jard, Diagnosis of asynchronous discrete event systems, a net unfolding approach, *IEEE Trans. on Automatic Control,* vol. 48, no. 5, pp. 714-727, May 2003.

[7] R.K. Boel, J.H. van Schuppen, Decentralized Failure Diagnosis for Discrete Event Systems with Costly Communication between Diagnosers, in Proc. 6th Int. Workshop on Discrete Event Systems, WODES'02, pp. 175-181, 2002.

[8] R.K. Boel, G. Jiroveanu, Distributed Contextual Diagnosis for very Large Systems, in Proc. of WODES'04, pp. 343-348, 2004.

[9] T. Chatain, C. Jard. Time Supervision of Concurrent Systems using Symbolic Unfoldings of Time Petri Nets, in: 3rd International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2005), Springer Verlag, September 2005, LNCS 3829, p. 196–210.

[10] O. Contant, S. Lafortune, Diagnosis of Modular Discrete Event Systems, in Proc. of WODES'04, pp. 337-342, 2004

[11] R. Debouk, S. Lafortune, D. Teneketzis, Coordinated Decentralized Protocols for Failure Diagnosis of Discrete Event Systems, *J. Discrete Event Dynamic Systems,* vol. 10(1/2), pp. 33-86, 2000.

[12] R. Debouk, S. Lafortune, and D. Teneketzis. On The Effect Of Communication Delays In Failure Diagnosis Of Decentralized Discrete Event Systems. Proc. of IEEE Conf. on Decision and Control, Sydney , Australia , December 12-15, 2000.

[13] Christophe Dousson, Paul Gaborit, Malik Ghallab: Situation Recognition: Representation and Algorithms. IJCAI 1993: 166-174

[14] R. Devillers and H. Klaudel. Solving Petri Net Recursions Through Finite Representation. Proc of IASTED'04.

[15] E. Fabre, Factorization of Unfoldings for Distributed Tile Systems, Part 1 : Limited Interaction Case, Inria research report no. 4829, April 2003. http://www.inria.fr/rrrt/rr-4829.html

[16] E. Fabre, Factorization of Unfoldings for Distributed Tile Systems, Part 2 : General Case, Inria research report no. 5186, May 2004. http://www.inria.fr/rrrt/rr-5186.html

[17] E. Fabre, Convergence of the turbo algorithm for systems defined by local constraints, Irisa research report no. PI 1510, 2003. http://www.irisa.fr/doccenter/publis/PI/2003/irisapublication.2006-01-27.8249793876

[18] E. Fabre, A. Benveniste, S. Haar, C. Jard, Distributed Monitoring of Concurrent and Asynchronous Systems, *J. Discrete Event Dynamic Systems,* special issue, vol. 15 no. 1, pp. 33-84, March 2005.

[19] E. Fabre, Distributed diagnosis based on trellis processes, in Proc. Conf. on Decision and Control, Sevilla, Dec. 2005, pp. 6329-6334.

[20] E. Fabre, C. Hadjicostis. A trellis notion for distributed system diagnosis with sequential semantics. In Proc. of Wodes 2006, Ann Arbor, USA, July 10-12, 2006.

[21] E. Fabre. Habilitation thesis. Uni. Rennes I. June 2007.

[22] C.J. Fidge. Logical time in distributed computing systems. *IEEE Computer* **24**(8), 28–33, 1991.

[23] S. Genc, S. Lafortune, Distributed Diagnosis Of Discrete-Event Systems Using Petri Nets, in proc. 24th Int. Conf. on Applications and Theory of Petri Nets, LNCS 2679, pp. 316-336, June, 2003.

[24] S. Genc and S. Lafortune. Distributed Diagnosis of Place-Bordered Petri Nets. *IEEE Transactions on Automation Science and Engineering*, January, 2007.

[25] S. Haar, A. Benveniste, E. Fabre, C. Jard. Fault Diagnosis for Distributed Asynchronous Dynamically Reconfigured Discrete Event Systems, in: IFAC World Congress Praha 2005, 2005.

[26] T. Jéron, H. Marchand, S. Pinchinat, and M-O. Cordier. Supervision Patterns in Discrete Event Systems Diagnosis. 8th International Workshop on Discrete Event Systems (Ann Arbor, Michigan, USA, 10-12 July 2006).

[27] R. Kumar and S. Takai. Inference-based Ambiguity Management in Decentralized Decision Making: Decentralized Diagnosis of Discrete Event Systems, 2006 American Control Conference, Minneapolis, June 2006.

[28] G. Lamperti and M. Zanella. Diagnosis of discrete-event systems from uncertain temporal observations. *Artif. Intell.* 137(1-2): 91-163 (2002).

[29] G. Lamperti and M. Zanella. *Diagnosis of Active Systems: Principles and Techniques.* Kluwer International Series in Engineering and Computer Science, Vol. 741, 2003.

[30] G. Lamperti and M. Zanella. Flexible diagnosis of discrete-event systems by similarity-based reasoning techniques. *Artif. Intell.* 170(3): 232-297 (2006).

[31] G. Lamperti and M. Zanella. Incremental Processing of Temporal Observations in Supervision and Diagnosis of Discrete-Event Systems. ICEIS (2) 2006: 47-57.

[32] S.L. Lauritzen. *Graphical Models.* Oxford Statistical Science Series 17, Oxford Univ. Press, 1996.

[33] S. Mac Lane. *Categories for the Working Mathematician.* Springer Verlag, 1998.

[34] K.L. McMillan, Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits, in Proc. 4th Workshop of Computer Aided Verification, Montreal, 1992, pp. 164-174.

[35] K.L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, PhD thesis, Kluwer, 1993.

[36] F. Mattern. Virtual time and global states of distributed systems, Proc. Int. Workshop on Parallel and Distributed Algorithms Bonas, France, Oct. 1988, Cosnard, Quinton, Raynal, and Robert Eds., North Holland, 1989.

[37] M. Nielsen, G. Plotkin, G. Winskel, Petri nets, event structures and domains, Theoretical Computer Science 13(1), 1981, pp. 85-108.

[38] J. Pearl. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence,* 29, 241-288, 1986.

[39] Y. Pencole, M-O. Cordier, L. Roze, A decentralized model-based diagnostic tool for complex systems. Int. J. on Artif. Intel. Tools, World Scientific Publishing Comp., vol. 11(3), pp. 327-346, 2002.

[40] W. Qiu and R. Kumar. Decentralized failure diagnosis of discrete event systems, *IEEE Transactions on Systems, Man & Cybernetics Part A,* pages 384-395, volume 36, number 2, 2006.

[41] W. Qiu and R. Kumar. A New Protocol for Distributed Diagnosis, 2006 American Control Conference, Minneapolis, June 2006.

[42] H.E. Rauch, F. Tung, and C.T. Striebel. Maximum Likelihood Estimates of Linear systems. *AIAA Journal,* (3), 1445-1450, Aug. 1965.

[43] M. Raynal, Distributed algorithms and protocols, Wiley & Sons, 1988.

[44] G. Rozenberg (ed.) *Handbook on Graph Grammars and Computing by Graph Transformation 1 (Foundations),* World Scientific, 1997.

[45] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, D. Teneketzis, Diagnosability of Discrete-event systems, *IEEE Trans. Autom. Control,* vol. 40(9), pp. 1555-1575, 1995.

[46] R. Su, Distributed Diagnosis for Discrete-Event Systems, PhD Thesis, Dept. of Elec. and Comp. Eng., Univ. of Toronto, June 2004.

[47] R. Su, W.M. Wonham, J. Kurien, X. Koutsoukos, Distributed Diagnosis for Qualitative Systems, in Proc. 6th Int. Workshop on Discrete Event Systems, WODES'02, pp. 169-174, 2002.

[48] R. Su, W.M. Wonham, Hierarchical Fault Diagnosis for Discrete-Event Systems under Global Consistency, *J. Discrete Event Dynamic Systems,* vol. 16(1), pp. 39-70, Jan. 2006.

[49] S. Tripakis. Undecidable Problems in Decentralized Observation and Control for Regular Languages. In *Information Processing Letters,* Volume 90, Issue 1, Pages 21-28 (15 April 2004).

[50] T. Yoo, S. Lafortune, A General Architecture for Decentralized Supervisory Control of Discrete-Event Systems, *J. Discrete Event Dynamic Systems*, vol. 12(3), pp. 335-377, July, 2002.

[51] Yin Wang, Stephane Lafortune, Tae-Sic Yoo. Decentralized Diagnosis of Discrete Event Systems Using Unconditional and Conditional Decisions. Proc. of the 44th IEEE Conference on Decision and Control Sevilla, Spain, December, 12-15, 2005.

[52] G. Winskel, Categories of models for concurrency, Seminar on Concurrency, Carnegie-Mellon Univ. (July 1984), LNCS 197, pp. 246-267, 1985.

[53] G. Winskel, Petri Nets, Algebras, Morphisms, and Compositionality , Information and Computation, no. 72, pp. 197-238, 1997.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399