CDC00-REG1625

Distributed State Reconstruction for Discrete Event Systems[1]

Eric Fabre, Albert Benveniste, Claude Jard,
Laurie Ricker, Mark Smith[2]

March 1, 2000

Abstract

We consider a discrete event dynamic system (DEDS) obtained by the parallel composition of several subsystems. Each subsystem can be seen as a standard stochastic DEDS. The composed system is provided with true concurrency semantics that emphasize concurrent behaviors of the subsystems. For these semantics, a trajectory appears as a partial order of events. For simplicity, we focus on the case of a global system composed of only two subsystems. We assume that firings in each subsystem are collected by a local sensor, which yields a sequence of transition labels (or events). The objective is to recover the most likely global trajectory of the system from the two (asynchronous) sequences of observations. This is an almost standard hidden state estimation problem, amenable to the clasical Viterbi algorithm. We propose a solution in which this global trajectory is built recursively by two asynchronously cooperating "players," each one being in charge of one subsystem. These two players run local Viterbi algorithms based on local states of the subsystems, plus some coordination information. This supervising architecture is particularly suited to large modular systems and is currently being applied to the distributed monitoring (and fault diagnosis) of telecommunication networks.

[1]This work is partially supported by RNRT (National Research Network in Telecommunication) through the Magda project (Modelling and Learning for a Distributed Management of Alarms). See `http://magda.elibel.tm.fr/` for extra information on the project.

[2]E.F. and A.B. are with IRISA-INRIA, C.J. is with IRISA-CNRS, L.R. is supported by an ERCIM fellowship, M.S. is supported by a MENRT fellowship ; Address : IRISA, Campus de Beaulieu, 35042 Rennes cedex, France ; E-mail : name@irisa.fr.

Contents

1 Introduction

This paper is motivated by the problem of fault diagnosis in distributed systems. A target application is the case of telecommunication networks[3], that are typically made up of many interconnected software and hardware components. In such systems, faults appearing in one component may propagate both in the network and in the protocol layers, and induce distant malfunctions. Each malfunction generally raises so-called "alarms" that are collected at different locations in the network. In today systems, human operators are drowned by alarms and hardly manage to visually "correlate" them in order to recover the original failure. Furthermore, identifying the alarm pattern resulting from a single failure is made difficult by the absence of a global time in the system : concurrent events may appear in different orders for the several occurences of the same original failure. In case of multiple faults, incredibly complex interleavings of alarm patterns may thus arise, which makes it impossible for a human to process the alarms.

Relying on this example, fault diagnosis can be seen as a state estimation problem, or more generally as a history reconstruction problem, where "history" denotes a sequence of states or events in the system. We assume that providing the operator with the successive propagations of faults is enough to characterize a failure. Such a reconstruction is driven by the alarms that the system produces as it evolves, and assumes that a model of the supervised system is available, which describes both its normal and faulty behavior. Classical solutions exist in the literature for general stochastic discrete event systems (DES), the generic one being the Viterbi algorithm (VA) for hidden Markov models [15]. Other solutions propose an off-line transform of the model (for non stochastic DES), the result being a deterministic system that directly produces state estimates when driven by observations. Such systems are called observers or diagnosers (the latter being direct extensions of observers) [5]. However, these solutions are faced with the curse of dimensionality : they handle explicitly enumarated states of the system, which makes them inapplicable to systems like telecommunication networks where the number of possible states is enormous. By contrast, a suitable supervising algorithm/structure should rely on the modular nature of the distributed system, which is typically obtained as the composition of subsystems. Ideally, one would imagine a structure of, say, communicating supervisors that would parallel the structure of the system itself, each supervisor being in charge of one subsystem, and relying on a local notion of state. Such a design has the advantage of being easy to upgrade as the structure of the distributed system evolves.

Several attempts have been made in this direction. An off-line approach is proposed in [10, 11], that builds a diagnoser for a system observed by several sensors. However, the local diagnosers in charge of processing their own sequence of observations handle states of the global model. Also, they know what are the transitions supervised by the other diagnosers, which determine coordination points. This solu-

[3]See the MADGA project, `http://maga.elibel.tm.fr`

tion still suffers from a dimensionality problem. A closer solution to what we present here is proposed in [12, 13]. It is an on-line solution, that obeys the observation-guided simulation principle of the Viterbi algorithm. The system model is defined as the composition of (non-stochastic) communicating automata, which is close to what we use here: we have chosen to replace communication buffers by automata that we call "shared resources." Subsystem (or "component") transitions are observed by local sensors. Each sensor processes its own sequence of observations, relying on the local supervised model, and assuming all possible interactions with neighboring subsystems. When local histories explaining local observations have been computed, they are merged into a global trajectory by using the constraint of coherent message circulation between subsystems. This framework is quite similar to the one we adopt in the present paper, and in particular relies on the same partial order semantics of events (although it is not mentioned explicitly). The extensions we propose assume – apart from the stochastic character – that observations are collected on the fly. In other words, instead of first computing local histories and then merging them, we do both at the same time, which prevents the algorithm from stopping at some predefined stop-points in the observations. The data structure we provide for representing trajectories allows to make "local merges" easily, without computing a global trajectory. Finally, the diagnosis algorithm we obtain is made of completely asynchronous processes, and does not require a centralized supervisor, nor a centralized data structure.

The distributed Viterbi algorithm we propose in this paper is dedicated to a stochastic distributed system (it extends some results of [6, 7, 8], developed for Petri nets). Special semantics are designed for the trajectories of the system, in order to represent explicitly the concurrency of events in subsystems. These semantics are the key to reduce the curse of dimensionality, because it is precisely the amount of concurrency in a distributed system that drives the exponential growth of the number of states. The paper is organized as follows. Section 2 is dedicated to a revisiting of the classical Viterbi algorithm. Here we do not take into account the distributed nature of the system, but rather focus on the structure of the algorithm itself. We explain how it can be distributed on several players that cooperate to solve the same problem. In the last part of section 2, an extension is made to the case where several sensors collect the observations in the system. Section 3 is dedicated to a modular, distributed VA. The system is presented as the composition of two subsystems, and special semantics are chosen to represent the concurrency of events. The distributed VA we design is made of two modules (or "players") that supervise one subsystem each, handle local states, and cooperate through some coordination information.

2 Distributed VA based on a global model

In this section, we revisit the standard Viterbi algorithm (VA) in order to introduce the suitable data structure that will be extended for the case of modular automata. We also indicate principles to distribute this algorithm for several cooperating players.

2.1 System

The system $\mathcal{S}$ we consider is a finite state machine (S, T, L, s_0), where S denotes the set of states, T the set of transitions, L a set of transition labels and $s_0 \in S$ the initial state of $\mathcal{S}$. A transition $t \in T$ is a triple (s^-, l, s^+) where s^- is the state enabling t, s^+ the state resulting from the firing of t, and l a label in L. Taking the label of t will be denoted by $L(t)$, and $S^-(t), S^+(t)$ will represent the initial and final states in t.

We call an *event* e the firing of some transition t in system $\mathcal{S}$. Formally, $T(e)$ will denote the transition fired in event e, but, in an abuse of notation, we may not always distinguish between transitions and events in the sequel. In this section, we assume that a *trajectory* τ is a finite *sequence of connectable events* $(e_1, e_2, \ldots, e_n)$ rooted at the initial state s_0. Naturally, two events e and e' are connectable (in this order) if they correspond respectively to the firing of transitions $t = (s^-, l, s^+)$ and $t' = (s'^-, l', s'^+)$ with $s^+ = s'^-$. The functions S^-, S^+ and L extend naturally to events and trajectories.

As usual, we assume the firing of t produces the emission of its label l, that we call an *observation* or *message*. The trajectory τ thus generates the sequence of observations $L(\tau) = (L(e_1), \ldots, L(e_n))$ that we shall denote by $(o_1, \ldots, o_n)$ or $o_1 \ldots o_n$. We assume $\mathcal{S}$ is a non-deterministic automaton, therefore the mapping $\tau \to L(\tau)$ is many-to-one. The objective of the Viterbi algorithm is precisely to invert this mapping : an observed sequence of labels is given, and one wishes to recover (recursively) the sequences of transition firings that could have produced it. We shall say that such sequences are *compatible* with observations, denoted by $\tau \sim o$, and $e_i \sim o_i$ for events.

In the case of a *stochastic* system, each transition t is assigned a likelihood $\mathbb{P}(t)$. The latter sums to one over transitions rooted at the same state s (provided there are such transitions) :

$$\forall s \in S, \qquad \sum_{t \in T \,:\, S^-(t)=s} \mathbb{P}(t) = 1$$

$\mathbb{P}$ encodes both the probability of changing state s into s' (the so-called transition probability), and also the probability of producing label l, given that this change has occured (the so-called observation probability). The likelihood $\mathbb{P}(\tau)$ of trajectory τ is the product of transition likelihoods over events appearing in τ. It is often more convenient to consider $-\log \mathbb{P}(t)$ that we call the *cost* $C(t)$ of transition t. The objective of the Viterbi algorithm is then to select the most likely trajectory τ, or equivalently the minimal cost trajectory, among those that match the observation sequence.

2.2 Representation of trajectories

In the algorithms we describe below, trajectories are represented by means of handles that we call *hooks*. The hook h representing τ is a tuple $h = (i, x, c, t, p)$ composed of

1. an index i, corresponding both to the number of events in τ and to the number of observations taken into account ; τ is compatible with $o_1 \dots o_i$,
2. a state x, corresponding to the final state of the τ, i.e. $S^+(\tau)$,
3. a cost c, corresponding to the cost $C(\tau)$ of the trajectory,
4. a transition t, corresponding to the last event connected to τ,
5. and a pointer[4] p towards a predecessor hook with index $i-1$, corresponding to the prefix τ' on which t was connected to yield τ.

We denote by I, X, C, T, P the functions that extract the above elements in a hook. With this definition, each hook h is equivalent to a complete trajectory τ. The latter can be recovered by recursively going through the predecessors of h : $P(h), P^2(h)$, etc., and collecting the sequence of events. τ is obtained by connecting the transitions encountered in this sequence :

$$\tau(h) = \Big(T(P^{I(h)-1}(h)) \ , \ \cdots \ , \ T(P^2(h)) \ , \ T(P(h)) \ , \ T(h) \Big)$$

This construction is usually called the "backtrack" in the Viterbi algorithm. It raises no algorithmic difficulty, so we shall not detail it any more in the sequel.

Observe that such a representation of trajectories corresponds to a *branching process*, as defined in [3] for example. This structure can be built recursively, which is the backbone of the Viterbi algorithm. By constrast with [3] however, we are going to *merge* some trajectories. If we are interested in the most likely trajectories, then this merge will actually correspond to a *selection* (branches will die out in the branching process).

2.3 Centralized VA

Two tasks must be achieved :

1. finding trajectories of $\mathcal{S}$ that are compatible with observations, and
2. selecting those with the lowest cost.

Both can be done recursively. For task 1, a guided simulation principle is applied : one tries to extend trajectories that are compatible with the first k observations by connecting one extra transition compatible with o_{k+1}. For task 2, one only has to

[4]For clarity, we will denote this pointer under the form of another hook h', but it must be understood that we actually mean "the address" of that hook, not the object itself.

notice that two trajectories that have produced the same observations o_1^k and that finish in the same state x will necessarily have identical extensions. Therefore, if one has a greater cost than the other, it will not be able to make up the cost difference. In other words, this trajectory can only be extended into non-optimal trajectories, and so can be removed from the search space. These two operations correspond to the *extension* and *reduction* steps defined below.

Extension. Let $\mathcal{A}$ denote a set of hooks, the *extension* of $\mathcal{A}$ is defined by

$$\mathsf{Ext}(h) = \left\{ \ (I(h)+1, S^+(t), C(h)+C(t), t, h) \ : \ t \in T,\ t \sim o_{I(h)+1},\ X(h) = S^-(t) \ \right\}$$

$$\mathsf{Ext}(\mathcal{A}) = \bigcup_{h \in \mathcal{A}} \mathsf{Ext}(h)$$

Hooks of $\mathcal{A}$ are thus augmented by one transition. Observe that *all* transitions of T are tested for the extension of each h. One only keeps transitions that are compatible with the next observation, and of course that are connectable to h.

Reduction. This corresponds to the elimination of trajectories that will not lead to optimal solutions. It is defined by

$$\mathsf{Red}(\mathcal{A}) = \left\{ h \in \mathcal{A} \ / \ \nexists h' \in \mathcal{A} : I(h) = I(h'),\ X(h) = X(h'),\ C(h) > C(h') \right\}$$

If several hooks have same state, same index and same optimal cost, all are kept[5].

Algorithm 1 Centralized VA

1. initialization

   a. $\mathcal{A} := \{(0, s_0, 0, \emptyset, \emptyset)\}$

2. forward recursion[6], until all hooks in $\mathcal{A}$ have index N

   a. select a non empty $\mathcal{A}' \subset \mathcal{A}$ of "unfinished" hooks

   b. $\mathcal{N} := \mathsf{Ext}(\mathcal{A}')$

   c. $\mathcal{A}'' := \mathsf{Red}(\mathcal{N})$

   d. $\mathcal{A} := (\mathcal{A} \setminus \mathcal{A}') \oplus \mathcal{A}''$

3. backtrack from the most likely hook(s) in $\mathcal{A}$

[5]They can be collapsed into a single hook with several predecessors, in order to reduce computations in future extensions.

[6]Hooks that are removed from the active set must be stored somewhere, since some of them will be necessary to perform the backtrack. We do not represent this operation but rather, assume that a hook is removed from the memory as soon as it cannot be reached by backtracking from the active set. This functionality is provided for free by some object-oriented programming languages (such as Java for example).

$\mathcal{A}$ is the set of "active hooks," which represents extremities of trajectories that can be extended. A trajectory is "finished" when it has reached the last observation o_N. At step 2.d, $\mathcal{U} \oplus \mathcal{V}$ denotes $\mathsf{Red}(\mathcal{U} \cup \mathcal{V})$. Since $\mathcal{U}$ and $\mathcal{V}$ are already reduced, the selection only has to compare elements of $\mathcal{U}$ to those of $\mathcal{V}$. The backtrack is not detailed ; it is initiated at the most likely hook(s) in $\mathcal{A}$.

Notice that the usual VA extends all current histories of $\mathcal{A}$ at each step. We present here a skew-synchronized version in order to prepare for the decentralized case, where extensions/reductions are performed in an asynchronous manner by several agents.

2.4 Decentralized VA : two players and a supervisor

This is a first version of a distributed computations algorithm. It is based on the observation that the extension/reduction step can be performed independently on several sets of active hooks, at the same time. We can therefore distribute this job to several processors (or *players*). We consider the case of two players here, where both know the global model $\mathcal{S}$ and the complete sequence of observations o_1^N. The term "decentralized" comes from the fact that a supervisor is in charge of distributing hooks to be processed, and of collecting results. Notation : i represents a generic player index ($i = 1, 2$), j denotes the other player index. For simplicity, we assume that communication channels between the supervisor and the players are FIFO and lossless[7].

Algorithm 2 : two players + one supervisor

- player i, $(i = 1, 2)$
  1. wait for start
  2. repeat until stopped by the supervisor
     a. get a set $\mathcal{A}'_i$ of active hooks from the supervisor
     b. compute $\mathcal{A}''_i := \mathsf{Vit}^{k_i}(\mathcal{A}'_i)$ for some integer k_i
     c. send back $\mathcal{A}''_i$ to the supervisor
- supervisor
  1. initialization
     a. $\mathcal{A} := \{(0, s_0, 0, \emptyset, \emptyset)\}$
     b. $\mathcal{A}'_1 := \emptyset$, $\mathcal{A}'_2 := \emptyset$
     c. start players 1 and 2
  2. repeat until all hooks in $\mathcal{A}$ have index N
     a. on request of player i, send $\mathcal{A}'_i \subset (\mathcal{A} \setminus \mathcal{A}'_j)$
     (a set of unfinished hooks)

[7]These assumptions can be relaxed by adopting more elaborate communication protocols.

b. on reception of a $\mathcal{A}''_i$ from player i, build $\mathcal{A} := (\mathcal{A} \setminus \mathcal{A}'_i) \oplus \mathcal{A}''_i$

3. stop players 1 and 2 (and start backtrack)

At step 2.b, players perform several Viterbi steps on the hooks they have received: $\mathsf{Vit}(\mathcal{A})$ denotes steps 2.a to 2.d in algorithm 1, and Vit^k denotes k successive steps of Vit. Notice that hooks are stored by the supervisor; the players are only in charge of computations.

2.5 Distributed VA: two players alone

By contrast with the decentralized case, the truly distributed case assumes that the supervisor has been removed. As a consequence, the players have to cooperate to achieve the three tasks performed by the supervisor, namely

1. distributing hooks to be processed,
2. collecting and storing the results, and
3. detecting that the work is finished.

It is quite easy for one player to receive hooks from the other player, compare them to/combine them with its own set of hooks, process local active hooks and send back new hooks to the other player (we specify below a policy for assigning hooks to players). Thus tasks 1 and 2 can be easily distributed. It is also easy to check *locally* that the job is finished. The major difficulty lies in the identification of the *global* end of the algorithm. The global termination point can be characterized by the fact that all players have finished, and that communication channels are empty[8]. This is exactly a *global termination detection problem*, a classic problem in distributed computing and for which several protocols are available. Therefore we assume that such a service is developed on top of local computations, which makes use of the boolean `local-end` and gives access to the boolean `global-end` in all players. Details can be found in [1] (chapter 4) or [2]. For simplicity, FIFO and lossless communication channels are still assumed.

Last question to solve: how to assign the processing of hooks to players. This was hidden inside the supervisor policy in the previous algorithm. For simplicity, we can assume that a given hook will have to be processed by player i iff it is of *type* i. The type of hook h is assumed to be a function of the index $I(h)$ and of the state[9] $X(h)$. Example: one can partition transition labels $L = L_1 \cup L_2$ and decide that player i will process labels L_i. Hence h is of type i if the next observation for h, $o_{I(h)}$, is a label of L_i. We denote by $\mathsf{Type}_i(\mathcal{A})$ the selection of hooks of type i in $\mathcal{A}$.

[8] Checking that communication channels are empty is the difficult point in general.

[9] The state can be augmented to account for a more elaborate definition of the type.

Algorithm 3 : two players (player i described)

1. initialization

   a. $\mathcal{A} := \{(0, s_0, 0, \emptyset, \emptyset)\}$

   b. $\mathcal{A}_i := \mathsf{Type}_i(\mathcal{A})$

2. forward sweep : until `global-end` is detected

   a. on decision to process local hooks

      – $\mathcal{A}'_i := \mathcal{A}_i$ (memory)

      – $\mathcal{A}_i := \mathsf{Vit}_i^{k_i}(\mathcal{A}_i)$ for some integer k_i

      – send $\mathsf{Type}_j(\mathcal{A}_i \setminus \mathcal{A}'_i)$ to player j (if non empty)

   b. on reception of a message $\mathcal{M}$ from player j

      – $\mathcal{A}_i := \mathcal{A}_i \oplus \mathcal{M}$

   c. `local-end` := all hooks in $\mathcal{A}_i$ are finished

3. backtrack

The subscript i in Vit_i indicates that only hooks of type i can be selected for extension. However, extending hooks of type i generally produces new hooks of both types, whence the third step of 2.a. Notice that step 2.a sends newly-created hooks, in order not to send back hooks previously received in some $\mathcal{M}$, nor hooks that were previously sent.

The backtrack is more complicated in this situation because the set of active hooks is divided between the two players. Therefore they have to agree on who has the best final hook, which can be done by broadcasting the best local likelihood. Notice that the backtrack needs to go from one player to the other, since the predecessor of a hook may be owned by the other player[10].

2.6 Multisensor distributed VA

Here we examine a first case where the players have only some partial knowledge of the global system. Let us assume that the set of transitions is partitioned into two subsets : $T = T_1 \cup T_2$, and $T_1 \cap T_2 = \emptyset$. This partition induces a decomposition of the set of states of system $\mathcal{S}$: $S = S_1 \cup S_2$, where S_i contains states involved in at least one transition of T_i (either as starting state or as resulting state). We proceed similarly for labels. This defines a decomposition of $\mathcal{S}$ into two automata $\mathcal{S}_i = (S_i, T_i, L_i, s_0)$ that share some states : $S_{1,2} = S_1 \cap S_2$. We assume player i knows only $\mathcal{S}_i$.

The trajectory semantics remain unchanged, but we assume that observations of T_1 and T_2 are collected by different "sensors." So each player only observes labels

[10]This distributed memory architecture also raises some difficulties for garbage collection, but we do not get into these details here.

stemming from the transitions he knows (see fig. 1). We thus have two sequences of observations : $o_1 \ldots o_{N_1}$ and $o'_1 \ldots o'_{N_2}$. This observation architecture is weaker than the centralized one because the interleaving of the two sequences is lost. Therefore, by contrast with the standard Viterbi problem, we have to recover the best trajectory matching *every possible interleaving of observations.*

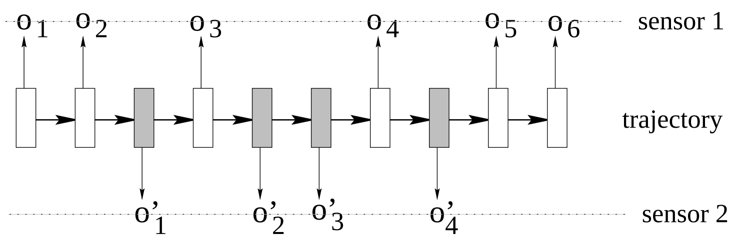

Figure 1: *A trajectory, viewed as a total order of events (represented by blocks). The color determines what set the corresponding transition belongs to. Each set is observed by a dedicated sensor.*

Going across all possible interleavings can be done in a simple way, by replacing the index i in hooks by a *double index* (i, j). This will indicate a hook compatible with an interleaving of $o_1 \ldots o_i$ and $o'_1 \ldots o'_j$. Extensions will thus augment i or j, according to the set in which the new transition is taken. In the same way, the reduction will select the most likely hook among those having the same state and the same double index. This defines a 2-dimensional recursion that contrasts with the 1-dimensional recursion of the standard VA. It represents the overhead one has to pay for checking all possible interleavings.

The distribution of the algorithm on the two players is slightly different from the previous case. Players only know part of the transitions. But, by definition, the extension must try to connect all possible transitions on a given hook. We thus decompose the extension in two extensions : one for each transition set T_i. As a consequence, hooks must be processed by *both* players. Therefore, in algorithm 3, as soon as a new hook is created, it must be communicated to the other player. We describe it in algorithm 4.

Algorithm 4 : two players with partial knowledge of $\mathcal{S}$ (player i described)

1. initialization

   a. $\mathcal{A}_i := \{(0, s_0, 0, \emptyset, \emptyset)\}$

   b. $\mathcal{E} := \emptyset$

2. forward sweep : until `global-end` is detected

   a. on decision of processing hooks

      – select $\mathcal{A}'_i \subset \mathcal{A}_i$

- $\mathcal{N}_i := \mathsf{Red} \circ \mathsf{Ext}_i(\mathcal{A}'_i)$
- $\mathcal{A}_i := (\mathcal{A}_i \setminus \mathcal{A}'_i) \oplus \mathcal{N}_i$
- $\mathcal{E} := \mathcal{E} \oplus \mathcal{N}_i$

b. on reception of message $\mathcal{M}$ from player j

- $\mathcal{A}_i := \mathcal{A}_i \oplus \mathcal{M}$

c. on decision of sending a message to player j

- if non-empty, send $\mathcal{E}$ to player j, $\mathcal{E} := \emptyset$

d. `local-end` = all hooks in $\mathcal{A}_i$ are finished

3. backtrack

Ext_i denotes the extension with transitions of T_i. The $\mathcal{E}$ set collects hooks created by each extension and stores them for a future emission to the other player. The backtrack is similar to algorithm 3.

Remark : When the interleaving of the observed sequences is known, it is possible to determine which player has to extend a given hook (i.e., hooks have a type). So algorithm 3 would be applicable.

3 Distributed VA for subsystems interacting by shared resources

As a first attemp to design a distributed VA, algorithm 4 already shows that it is possible to handle distributed observations, and to combine local VA working with some partial knowledge of the global system. However, this algorithm suffers from two major drawbacks :

1. In large distributed systems (a telecommunication network for example), it is impossible to handle the global state of the system. Unfortunately, algorithm 4 is based on global states.

2. Trying to find the most likely trajectory under the form of a *sequence* of events requires a huge overhead in computation time. This information is often meaningless in a distributed system, where the exact ordering of concurrent events has no influence on the behavior of the system.

The solution to point 2 consists in adopting *partial order semantics* for trajectories. These semantics were originally developed for Petri nets [4, 6] but can be easily extended to standard finite state machines, as presented in [3], for example. To do so, instead of decomposing $\mathcal{S}$ into subsystems by partitioning its transition set, we construct $\mathcal{S}$ as the parallel composition of small systems. This also opens the way to a solution to point 1 : we are going to build a local VA for each player that handles states of its subsystem (plus some synchronization information) instead of global states of $\mathcal{S}$.

3.1 Systems interacting by shared resources

We model $\mathcal{S}$ as composed of two local subsystems $\mathcal{S}_A$ and $\mathcal{S}_B$ with their own state space and transition set : $\mathcal{S}_A = (A, T_A, L_A, a_0)$ and similarly for $\mathcal{S}_B$. The interaction between $\mathcal{S}_A$ and $\mathcal{S}_B$ is established by another system $\mathcal{S}_C = (C, T_C, L_C, c_0)$. Specifically, transition sets T_A and T_B are partitioned into *local* and *shared* transitions : $T_A = T_A^l \cup T_A^s$ (idem for T_B). Shared transitions are *synchronized* with transitions of $\mathcal{S}_C$ through the one to one mapping ϕ:

$$\phi \;:\; T_A^s \cup T_B^s \longrightarrow T_C$$

System $\mathcal{S}$ is thus obtained as the *parallel composition* $\mathcal{S} = \mathcal{S}_\mathcal{A} \| \mathcal{S}_\mathcal{C} \| \mathcal{S}_\mathcal{B}$ (see [5] p. 85, or the *synchronous product* in [3]). In this composition, states of $\mathcal{S}$ are triples (a, c, b), with (a_0, c_0, b_0) as initial state. Local transitions of T_A^l (resp. T_B^l) change only the a part (resp. b), but shared transitions of T_A^s (resp. T_B^s) are stuck with their counterpart by ϕ in $\mathcal{S}_C$, so that they change both a (resp. b) and c. In other words, $\mathcal{S}_A$ and $\mathcal{S}_B$ are in competition for the access to the common resource c (they cannot use it at the same time). This situation parallels the case of two Petri nets that share some places, as illustrated in figure 2. Pushing further the comparison with Petri nets, we say that two transitions are *structurally concurrent* iff they make use of different resources (i.e. modify different components of the global state (a, c, b)). Pairs of structurally concurrent transitions are given by $(T_A \times T_B) \setminus (T_A^s \times T_B^s)$.

3.2 Partial order semantics for trajectories

For the partial order semantics, trajectories are *partial orders ef events* instead of sequences. They are represented by a *directed acyclic graph (DAG)* of events, which is usually chosen to be the transitive reduction of the partial order (see the notion of *occurence net* in [3], the notion of *puzzle* in [6, 8], or, more generally, the notion of *unfolding* in [4]).

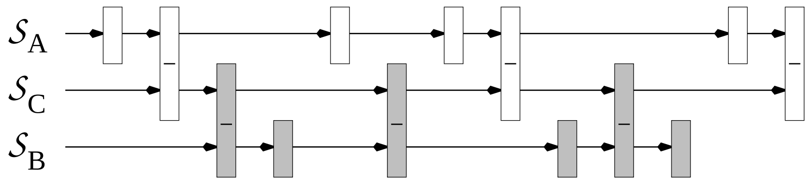

Figure 2: *Partial order semantics for a trajectory of $\mathcal{S}_A \| \mathcal{S}_C \| \mathcal{S}_B$. White blocks indicate transitions of T_A, grey blocks those of T_B. Small blocks represent local transitions (that operate on local states only), whereas long blocks represent shared transitions that synchronize with a transition of T_C and thus also change the shared state. The partial order semantics assume that transitions are totally ordered in each single automaton, but no order is specified between pairs of concurrent transitions of $\mathcal{S}_A$ and $\mathcal{S}_B$.*

Definition 1 *A DAG of events of $\mathcal{S}$ is a valid trajectory for the partial order semantics iff*

a) its restriction to events of $\mathcal{S}_A$ (and resp. $\mathcal{S}_B$, $\mathcal{S}_C$) is a valid sequence *of events for $\mathcal{S}_A$ (resp. $\mathcal{S}_B$, $\mathcal{S}_C$), and*

b) two events are ordered in the DAG iff they are ordered in one of the above restrictions.

We call such a DAG an event graph[11].

This definition is illustrated by figure 2. The partial order semantics are designed to represent explicitly the true concurrency of events : the order in which two concurrent transitions fire (provided they are simultaneously enabled) is left unspecified. As a direct consequence of the definition, every linear extension of an event graph is a valid sequence of events for $\mathcal{S}$, and yields the same final global state. It is precisely because some orderings are dropped that the overhead in algorithm 4 can be reduced.

3.3 Link with stochastic systems

The partial order semantics are not compatible with the usual definition of stochastic systems. The latter can only assign likelihoods to *sequences* of events, whereas the former does not distinguish sequences that are extensions of the same event graph. In other words, standard Markov dynamics for transition systems assign different likelihoods to different orderings of concurrent events, which is not desirable here. Moreover, likelihoods usually depend on the global state of the system, which is also something we want to avoid here.

To remove these drawbacks, a formalism has been designed in [6, 8] to establish a parallel between the concurrency of events and their stochastic independence. This can be achieved by giving up the idea of Markov dynamics, and by considering *partially randomized systems*. For the example we consider here, this amounts to considering $\mathcal{S}_A$ and $\mathcal{S}_B$ as standard independent stochastic systems, the trajectories of which are *constrained* by the presence of $\mathcal{S}_C$. The order in which $\mathcal{S}_A$ and $\mathcal{S}_B$ fire *is not randomized* but simply left unknown, whence the name of "partially randomized system[12]." In this setting, the likelihood of an event graph is obtained as the product of the sequence likelihoods in $\mathcal{S}_A$ and $\mathcal{S}_B$, up to a renormalization constant. $\mathcal{S}_C$ doesn't introduce any randomness ; it is used as a constraint on pairs of sequences of $\mathcal{S}_A$ and $\mathcal{S}_B$. Therefore the cost of transitions can be left unchanged, and the cost of event graphs is evaluated as in section 2.

[11] In another community, event graphs denote Petri nets with special properties. This has no connection with the structure we consider here.

[12] To establish ordinary Markov dynamics, one would have also to randomize the choice of the system that fires, when both are enabled. This brings back the two drawbacks mentioned above.

3.4 Representation of trajectories

Definition 1 expresses an event graph in terms of sequences. It is therefore natural to rely on the previous representation of sequences by hooks. We represent a trajectory for $\mathcal{S}$ by a hook h which is a triple of hooks of $\mathcal{S}_A, \mathcal{S}_C$ and $\mathcal{S}_B$: $h = (h_A, h_C, h_B)$. h_A (resp. h_B) is defined exactly like in section 2.2 ; in particular, its index measures the number of events in $\mathcal{S}_A$ (resp. $\mathcal{S}_B$), its cost corresponds to the cost in $\mathcal{S}_A$ (resp. $\mathcal{S}_B$) and the pointer indicates a previous hook for $\mathcal{S}_A$ (resp. $\mathcal{S}_B$). The structure of h_c is simpler : the cost is useless ($\mathcal{S}_C$ doesn't participate to the cost), and the index is useless also ($\mathcal{S}_C$ doesn't have proper transitions).

Extensions must be adapted to this new structure. Two cases must be considered for Ext_A, which denotes an extension by transitions of $\mathcal{S}_A$:

1. When a *local* transition of $\mathcal{S}_A$ is connected to the event graph represented by $h = (h_A, h_C, h_B)$, only the h_A hook is changed, exactly as it would be for $\mathcal{S}_A$ alone. This yields $h' = (h'_A, h_C, h_B)$ with h'_A pointing back towards h_A.
2. When a *global* transition t_A of $\mathcal{S}_A$ is connected to $h = (h_A, h_C, h_B)$, both h_A and h_C are changed, as they would be for $\mathcal{S}_A$ and $\mathcal{S}_C$ alone. This yields $h' = (h'_A, h'_C, h_B)$ with h'_A pointing back towards h_A, h'_C pointing back towards h_C, and h'_A, h'_C marked by synchronized transitions : t_A and $\phi(t_A)$ respectively.

The same holds for transitions of $\mathcal{S}_B$. From this structure, one easily checks that a hook h represents a unique event graph (for example, the verification can be done by backtracking from h ; this is left to the reader).

3.5 Separation property

We establish here the key result that makes possible a distributed algorithm working on local states. Let $h = (h_A, h_C, h_B)$ be a hook, representing an event graph of $\mathcal{S}$. We define the two canonical projections ψ_A and ψ_B by

$$\begin{aligned} \psi_A(h) &= (h_A, h_C) \\ \psi_B(h) &= (h_C, h_B) \end{aligned}$$

Definition 2 *Let $o = o_1 \dots o_{N_1}$ and $o' = o'_1 \dots o'_{N_2}$ be two sequences of observations on the transitions of $\mathcal{S}_A$ and $\mathcal{S}_B$ respectively. Let $h = (h_A, h_C, h_B)$ be a hook for $\mathcal{S}$. h is said to be compatible with o and o' iff h_A (and resp. h_B) represents a sequence of events in $\mathcal{S}_A$ (resp. $\mathcal{S}_B$) that is compatible with some prefix $o_1 \dots o_i$ for some integer $i \leq N$ (resp. $o'_1 \dots o'_j$ for some integer $j \leq N$).*

Proposition 1 *Let $\mathcal{A}$ be a set of hooks for $\mathcal{S}$ that are compatible with o and o'. Let $\mathcal{A}'$ be obtained by the product $\psi_A(\mathcal{A}) \otimes \psi_B(\mathcal{A})$ defined by*

$$\begin{aligned} \mathcal{A}' &= \psi_A(\mathcal{A}) \otimes \psi_B(\mathcal{A}) \\ &\triangleq \{(h_A, h_C, h'_B) \ : \ (h_A, h_C) \in \psi_A(\mathcal{A}), \ (h'_C, h'_B) \in \psi_B(\mathcal{A}), \ h_C = h'_C\} \end{aligned}$$

then all hooks in $\mathcal{A}'$ are compatible with o and o'.

Notice that $\mathcal{A} \subset \mathcal{A}'$, but $\mathcal{A}'$ may be bigger. This result states that two event graphs for $\mathcal{S}_A \| \mathcal{S}_C$ and $\mathcal{S}_C \| \mathcal{S}_B$ that coincide on $\mathcal{S}_C$ can be merged in an event graph for $\mathcal{S}_A \| \mathcal{S}_C \| \mathcal{S}_B$ (report to colors on figure 2). In other words, given that they agree on the interleaving of shared transitions, trajectories in $\mathcal{S}_A$ and $\mathcal{S}_B$ are "independent :" the "holes" between consecutive shared transitions can be filled by any - legal - sequence of local transitions.

Proof. One solution is to prove that if the property holds for $\mathcal{A}$, then it holds also for $\mathsf{Ext}_A(\mathcal{A})$, and thus for $\mathsf{Ext}_B(\mathcal{A})$ by symmetry. This point is quite straightforward. Since the property holds for the initial state, the induction already proves the result for all sets of hooks that can be reached by extension, which is enough for us.

The direct proof is easy. Let $h, h' \in \mathcal{A}$, then $\psi_A(h)$ is an event graph for $\mathcal{S}_A \| \mathcal{S}_C$ that is compatible with o, and $\psi_B(h')$ is an event graph for $\mathcal{S}_C \| \mathcal{S}_B$ that is compatible with o'. If they have the same trajectory on $\mathcal{S}_C$, i.e., the same h_C component, the trajectories represented by h_A, h'_B and h_C define a unique event graph, which is a valid trajectory for $\mathcal{S}$ (see definition 1).

3.6 Distributed algorithm

Relying on proposition 1, we represent the set of active hooks under a *product form*, i.e., through the one-to-one correspondence

$$\mathcal{A} \quad \longleftrightarrow \quad \psi_A(\mathcal{A}) \otimes \psi_B(\mathcal{A})$$

Naturally, the player observing transitions of $\mathcal{S}_A$ (denoted by "player A") will be in charge of the $\psi_A(\mathcal{A})$ part, and symmetrically "player B," observing $\mathcal{S}_B$, will handle the $\psi_B(\mathcal{A})$ part.

Extension. Extensions do not need to be redefined : Ext_A works on pairs of $\psi_A(\mathcal{A})$ – with transitions of $\mathcal{S}_A$ – exactly as explained in section 3.4 (idem for Ext_B).

Let $\mathcal{A}_A$ be a set of pairs (h_A, h_C) for player A. As in algorithm 4, a subset $\mathcal{A}'_A$ of $\mathcal{A}_A$ will be extended to give a set $\mathcal{N}_A$ of new pairs. We assume that each time a pair $(h_A, h_C) \in \mathcal{A}_A$ is extended into (h'_A, h'_C) by a shared transition, the corresponding *change* $h_C \to h'_C$ is stored in a set $\mathcal{C}_A$. In other words, $\mathcal{C}_A$ stores the image on $\mathcal{S}_C$ of the extension performed by player A. We include the storage of this subproduct in the notation :

$$(\mathcal{N}_A, \mathcal{C}_A) := \mathsf{Ext}_A(\mathcal{A}'_A)$$

We also need to keep track of components h_C that disappear from $\mathcal{A}_A$ after the extension. Referring to the update equation $\mathcal{A}_A := (\mathcal{A}_A \setminus \mathcal{A}'_A) \oplus \mathcal{N}_A$, the set of finished hooks h_C is defined by

$$\mathcal{F}_A = \mathsf{Fin}(\mathcal{A}'_A, \mathcal{N}_A, \mathcal{A}_A) = \psi_C(\mathcal{A}'_A) \setminus [\psi_C(\mathcal{N}_A) \cup \psi_C(\mathcal{A}_A \setminus \mathcal{A}'_A)]$$

where ψ_C denotes the canonical projection on the h_C component (in this equation, we naturally take $\mathcal{A}_A$ *before* it is updated).

Reduction. The reduction operation differs from what has been seen up to now since it is not performed on global hooks h but on their projections. Let $\mathcal{A}_A$ be a set of pairs (h_A, h_C)

$$\begin{aligned} \mathsf{Red}(\mathcal{A}_A) \;=\; \Big\{ & (h_A, h_C) \in \mathcal{A}_A \Big/ \nexists (h'_A, h'_C) \in \mathcal{A}_A \;:\; h_C = h'_C, \\ & I(h_A) = I(h'_A),\; X(h_A) = X(h'_A),\; C(h_A) > C(h'_A) \Big\} \end{aligned} \tag{1}$$

That is, the selection on histories of $\mathcal{S}_A$ (with identical lengths) is conditioned by the fact that they finish in the same state *and* that they are synchronized with the same *history* on $\mathcal{S}_C$. Which means that the notion of "state" for recursive optimisation in the local Viterbi is the pair $(X(h_A), h_C)$.

Let us briefly comment that, in general, the possibility of optimization "by parts" is precisely due to a factorisation result. In the case of the standard Viterbi algorithm, one relies on the Markov property : the cost of a sequence separates into cost of the past plus cost of the future, given the present, whence the recursive optimization. Here, by proposition 1, we have introduced a factorisation "in space" for trajectories of $\mathcal{S}$. Considering both factorisations, in time and space, one gets the criterion of (1).

Active hooks. As for algorithm 4, we are in a partial knowledge setting : each player knows only part of the transitions that could extend a given global hook $h = (h_A, h_C, h_B)$. So *both* have to process h. Since we store active hooks by their images through ψ_A and ψ_B, the double extension takes a special form. An active pair (h_A, h_C) in player A may be extended either directly, by an extension performed by player A, or undirectly, by the extension of one of its possible counterparts in player B. In this latter case, only the h_C component is changed. Therefore, active pairs (h_A, h_C) of player A are stored in two sets :

- $\mathcal{A}_A$ containing pairs that can be extended by player A with transitions of $\mathcal{S}_A$,
- $\mathcal{W}_A$ containing pairs already extended by player A, that *wait* for a possible extension of one of their companions in player B.

After a pair (h_A, h_C) has been extended by player A, it is removed from $\mathcal{A}_A$ and stored in $\mathcal{W}_A$. The reception of a message $h_C \to h'_C$ from player B indicates that some companion (h_C, h_B) has been extended in (h'_C, h'_B). So (h_A, h_C) can be *updated* into (h_A, h'_C) and put back in $\mathcal{A}_A$ for future extensions by player A. We denote by Update this operation. On the contrary, receiving a message "h_C is finished" indicates that player B has finished computing possible extensions for pairs $(h_C, \cdot)$, and (h_A, h_C) can therefore be removed from the set of waiting pairs.

Algorithm 5 two players, distributed knowledge (player A described)

1. initialization

a. $h_A^0 := (0, a_0, 0, \emptyset, \emptyset)$
b. $h_C^0 := (c_0, \emptyset, \emptyset)$
c. $\mathcal{A}_A := \{(h_A^0, h_C^0)\}$
d. $\mathcal{W}_A := \emptyset$

2. forward sweep : until `global-end` is detected

   a. on decision of processing hooks
      - select $\mathcal{A}'_A \subset \mathcal{A}_A$
      - $(\mathcal{N}_A, \mathcal{C}_A) := \mathsf{Ext}_A(\mathcal{A}'_A)$
      - $\mathcal{N}_A := \mathsf{Red}_A(\mathcal{N}_A)$
      - $\mathcal{F}_A := \mathsf{Fin}(\mathcal{A}'_A, \mathcal{N}_A, \mathcal{A}_A)$
      - $\mathcal{A}_A := (\mathcal{A}_A \setminus \mathcal{A}'_A) \oplus \mathcal{N}_A$
      - $\mathcal{W}_A := \mathcal{W}_A \oplus \mathcal{A}'_A$
      - send $(\mathcal{C}_A, \mathcal{F}_A)$ to player B

   b. on reception of message $(\mathcal{C}_B, \mathcal{F}_B)$ from player B
      - $\mathcal{A}_A := \mathcal{A}_A \oplus \mathsf{Update}(\mathcal{A}_A, \mathcal{C}_B) \oplus \mathsf{Update}(\mathcal{W}_A, \mathcal{C}_B)$
      - remove from $\mathcal{W}_A$ all pairs (h_A, h_C) such that $h_C \in \mathcal{F}_B$

   c. `local-end` = all hooks in $\mathcal{A}_A$ are finished

3. backtrack

The backtrack is more complicated than in algorithm 4 since it does not operate on global hooks, but it raises no severe difficulty.

Notice that the `send` operation is now included in the processing loop. This is to account for the fact that messages are *ordered*. Of course, it remains possible to keep messages in a FIFO buffer, and send them at any time.

The computation and processing of $\mathcal{F}_A, \mathcal{F}_B$ messages is only meant to save memory and avoid the explosion of the $\mathcal{W}$ sets. The algorithm remains valid without these operations.

Algorithm 5 offers the crucial advantage of working on local states of subsystems, regardless of the size of the global system. However, the interface of a player with the rest of the system relies on hooks h_C, which represent *histories* of $\mathcal{S}_C$. The number of possible interface histories may grow very rapidly, which is the price to pay for not handling global states of $\mathcal{S}$. This point remains to be checked, but one can suspect that a correct definition of local systems, that maximizes concurrency of $\mathcal{S}_A$ and $\mathcal{S}_B$, avoids this explosion.

4 Conclusion

We have presented a distributed state reconstruction algorithm for a discrete event system described as the parallel composition of subsystems. The structure of this algorithm parallels the structure of the system itself: an agent is designed for each subsystem. Each agent knows only its subsystem plus the interface with other subsystems. It observes only its local transitions, and runs a local VA that handles local states, plus some synchronization information with the rest of the system. This distributed algorithm is made possible by the use of partial order semantics for trajectories of the global system, which exploits explicitly the concurrency of events.

This algorithm is particularly suited to large concurrent systems, and offers an alternative to the curse of dimensionality. Several direct extensions are currently being investigated, in particular the generalization to N subsystems, which makes a nice connection with the so-called Bayesian Networks (the latter are fix-time structures). Other directions are also explored to make this approach applicable to real case problem. Let us mention for example the definition of distributed stop-points for the algorithm, the possibility of losing some observations, or the case of an incomplete knowledge of the system. An application of this algorithm is currently developed for the monitoring of telecommunication networks, in the framework of the MAGDA project.

References

[1] Michel Raynal, "Algorithmes distribués et protocoles," Eyrolles, 85.

[2] Jean-Michel Helary, Claude Jard, Noël Plouzeau, Michel Raynal, "Detection of stable properties in distributed applications," internal report no. 342, Irisa, Jan. 87.

[3] Javier Esparza, Stefan Römer, "An unfolding algorithm for synchronous products of transition systems," in proceedings of CONCUR'99, LNCS 1664.

[4] W. Vogler, "Modular Construction and Partial Order Semantics of Petri Nets," LNCS no. 625, 1992.

[5] Christos G. Cassandras, Stéphane Lafortune, "Introduction to discrete event systems," Kluwer Academic Publishers, 1999.

[6] A. Aghasaryan, E. Fabre, A. Benveniste, R. Boubour, C. Jard, "Fault Detection and Diagnosis in Distributed Systems : an Approach by Partially Stochastic Petri nets," Journal of Discrete Event Dynamic Systems, special issue on Hybrid Systems, vol. 8, pp. 203-231, June 98.

[7] R. Boubour, C. Jard, A. Aghasaryan, E. Fabre, A. Benveniste, "A Petri net approach to fault detection and diagnosis in distributed systems. Part I : application to telecommunication networks, motivations and modeling." CDC'97 Proceedings, San Diego, December 1997.

[8] A. Aghasaryan, E. Fabre, A. Benveniste, R. Boubour, C. Jard, "A Petri net approach to fault detection and diagnosis in distributed systems. Part II : extending Viterbi algorithm and HMM techniques to Petri nets." CDC'97 Proceedings, San Diego, December 1997.

[9] A. Benveniste, B.C. Levy, E. Fabre, P. Le Guernic, "A Calculus of Stochastic Systems : Specification, Simulation, and Hidden State Estimation," Theoretical Computer Science, no. 152, pp. 171-217, 1995.

[10] R. Debouk, S. Lafortune, D. Teneketzis, "A coordinated decentralized protocol for failure diagnosis of discrete event systems," Proc. of WODES'98, Cagliari, Italy, 1998.

[11] R. Debouk, S. Lafortune, D. Teneketzis, "Coordinated decentralized protocols for failure diagnosis of discrete event systems," Journal of Discrete Event Dynamic Systems : Theory and Applications, vol. 10, no. 1-2, pp. 33-86, Jan. 2000.

[12] P. Baroni, G. Lamperti, P. Pogliano, M. Zanella, "Diagnosis of active systems," Proc. of ECAI'98 (13th European Conf. on Artif. Intel.), 1998.

[13] P. Baroni, G. Lamperti, P. Pogliano, M. Zanella, "Diagnosis of large active systems," Artificial Intelligence 110, pp. 135-183, 1999.

[14] R. David, H. Alla, "Petri Nets for Modeling of Dynamic Systems - A Survey," Automatica, vol. 30, no. 2, pp. 175-202, 1994.

[15] L.R. Rabiner, "A tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," Proceedings of IEEE, vol. 77, no.2, February 1989.

[16] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, "Modeling with Generalized Stochastic Petri Nets," Wiley Series in Parallel Computing, 1995.

[17] M. Ajmone Marsan, G. Balbo, G. Chiola, G. Conte, "Generalized Stochastic Petri Nets Revisited: Random Switches and Priorities," in Proc. of PNPM '87, IEEE-CS Press, pp. 44-53.