Distributed State Reconstruction for Discrete Event Systems[1]

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

Abstract

We consider the state estimation problem for stochastic discrete event dynamic system (DEDS) obtained by the parallel composition of several subsystems. A distributed inference algorithm is developed in the case of distributed observations. It is composed of asynchronous agents that only have a local view of the model and of observations. This algorithm only handles local states of subsystems, which is a way of avoiding the state explosion difficulty of large concurrent systems.

1 Introduction

This work is motivated by the problem of fault diagnosis in distributed systems, with telecommunication networks as target application. Such systems are typically made up of many interconnected software and hardware components, and failures appearing in one component may propagate to the others, causing distant malfunctions. All impacted components raise so-called "alarms" that are collected by a supervisor. Discovering the original failure from an alarm pattern has become an impossible task for a human operator : the number of alarms is too large, and alarm patterns are too complex. This is due to the distributed nature of the system, where no notion of global time is available. The same original failure may thus produce different orderings of alarms, and in case of multiple failures, incredibly complex interleavings of alarm patterns arise.

Relying on this example, fault diagnosis appears as a standard history reconstruction problem. Assuming a model of the supervised system is available, which describes both its normal and faulty behavior, together with the production of observable events, one has to recover the most likely hidden trajectory of the model that explains observations. Many solutions exist in the literature for general stochastic discrete event systems, either offline (diagnosers [5]) or online (Viterbi algorithm (VA), for hidden Markov models [15]). However, these solutions are faced with the curse of dimensionality : they handle global states of the system, which makes them inapplicable to large distributed systems.

One would imagine instead a distributed diagnosis architecture that would parallel the structure of the system itself : one local supervisor being in charge of one subsystem, handling a local notion of state, and synchronizing its work with neighbour supervisors. Such a design has the advantage of being easy to upgrade as the distributed system evolves. Several attempts have been made in this direction [10, 11], but Baroni *et al.* [12, 13] have probably proposed the first solutions based on local states, in the formalism of communicating automata. A local diagnoser is defined on each component, and processes local observations, assuming all possible synchronization messages with neighbour components. Local solutions are then merged by the constraint of coherent message circulation between components. We follow similar ideas, but design a completely asynchronous on-line algorithm, where local merges can be done at any time, without reconstructing global states nor using a global supervisor.

The paper is organized as follows. Section 2 revisits the classical VA for a standard automaton and explains how it can be distributed on several agents and extended to the case of distributed observations. Section 3 is dedicated to a modular, distributed VA. The system is composed of two stochastic subsystems, and special semantics are chosen to represent the concurrency of events. The distributed VA is made of two agents, one per subsystem, which handle local states and cooperate through some coordination information.

2 Distributed VA based on a global model

In this section, we revisit the standard 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.

[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.

2.1 System

The system we consider is a finite state machine (FSM) $\mathcal{S} = (\mathcal{S}, \mathcal{T}, \mathcal{L}, \int_t)$, 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 is denoted by $L(t)$, and $S^-(t), S^+(t)$ represent the initial and final states in t.

We call an *event* e the firing of some transition t of $\mathcal{S}$ (in an abuse of notation, we may not always distinguish between transitions and events in the sequel). In this section, 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 firing transitions $t = (s^-, l, s^+)$ and $t' = (s'^-, l', s'^+)$ with $s^+ = s'^-$. 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*. A trajectory τ thus generates the sequence of observations $L(\tau) = (L(e_1), \ldots, L(e_n))$ that we denote by $o_1 \ldots o_n$. We assume $\mathcal{S}$ is a non-deterministic automaton, therefore the mapping $\tau \to L(\tau)$ is many-to-one. The objective of the VA is precisely to invert this mapping: an observed sequence of labels $o = o_1 \ldots o_N$ is given, and one wishes to recover (recursively) the sequences of transition firings that could have produced it. Such sequences are said to be *compatible* with observations, denoted by $\tau \sim o$, and $e_i \sim o_i$ for events.

We consider *stochastic* systems: each transition t is assigned a likelihood $\mathbb{P}(t)$. The latter sums to one over transitions rooted at the same state s:

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

$\mathbb{P}$ encodes both the transition probability and the observation probability, given some state change. 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 VA is then to select the most likely trajectory τ, or equivalently the minimal cost trajectory, in $L^{-1}(o)$.

2.2 Representation of trajectories

In the algorithms we describe below, trajectories are represented by means of a pointer structure. The building element is called a *hook*, denoted by h, which represents a complete trajectory τ. A hook is a 5-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 $o_1 \ldots o_i$,
2. a state x, corresponding to the final state $S^+(\tau)$,
3. a cost c, corresponding to the cost $C(\tau)$,
4. a transition t, corresponding to the last event connected to τ,
5. and a pointer[1] 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. The trajectory τ represented by h can be recovered by recursively going through the predecessors of h : $P(h), P^2(h)$, etc., and connecting the associated sequence of events (the so-called "backtrack" step of the VA).

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 VA. By constrast with [3] however, we are going to *merge* some trajectories. If we are interested in the most likely trajectories, 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 notice that two trajectories that have produced the same observations $o_1 \ldots o_k$ 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 $\mathsf{Ext}(\mathcal{A}) = \bigcup_{h \in \mathcal{A}} \mathsf{Ext}(h)$ where

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

Hooks of $\mathcal{A}$ are thus augmented by one event. Observe that *all* transitions of T are tested for the extension of each h.

[1] 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.

Reduction. It corresponds to the elimination of trajectories that will not lead to optimal solutions.

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

If several hooks have same state, same index and same optimal cost, all are kept[2].

Centralized VA :

1. initialization $h^0 = (0, s_0, 0, \emptyset, \emptyset)$, $\mathcal{A} := \{h^0\}$
2. forward sweep, until all hooks in $\mathcal{A}$ have index N
   a. select $\mathcal{A}' \subset \mathcal{A}$, a set of hooks with $I(h) < N$
   b. $\mathcal{A}'' := \mathsf{Red} \circ \mathsf{Ext}(\mathcal{A}')$
   c. $\mathcal{A} := (\mathcal{A} \setminus \mathcal{A}') \oplus \mathcal{A}''$
3. backtrack from the most likely hook(s) in $\mathcal{A}$

$\mathcal{A}$ is the set of "active hooks:" it stores extremities of trajectories that can be extended. At step 2.c, $\mathcal{U} \oplus \mathcal{V}$ denotes $\mathsf{Red}(\mathcal{U} \cup \mathcal{V})$. Since $\mathcal{U}$ and $\mathcal{V}$ are already reduced sets, the selection only has to compare elements of $\mathcal{U}$ to those of $\mathcal{V}$.

2.4 Distributed VA

The usual VA extends *all* current histories of $\mathcal{A}$ at each step, whence a recursion indexed by n, the number of observations taken into account. The skew-synchronized version presented above drops this assumption, which allows extensions/reductions to be performed in parallel by several asynchronous agents, or "players." Let us sketch first a *decentralized* architecture, composed of a supervisor and two (or more) players, linked to the supervisor by lossless FIFO channels. We assume each player knows the complete system model and the observed sequence of labels. The supervisor sends a subset of hooks to be processed to a player, on request, and removes them from the active set. After the player has performed some rounds of extensions/reductions on these hooks, the result is sent back to the supervisor that re-incorporates the new histories in its set of active hooks using the $\oplus$ operator. The players work independently of each other, and only communicate with the supervisor. The latter stops computations when all players have sent their results and the active set contains only "finished" hooks (i.e. hooks with N as index value). Then the supervisor initiates the backtrack with the most likely hook.

In this architecture, the supervisor doesn't know the model, nor the observations. His job is 1/ to distribute hooks to process to the players, 2/ to collect, compare and store the results of computations, which is a kind of blackboard function, and 3/ to detect that the work is finished. A truely *distributed* architecture goes further by removing the supervisor, which means that the three tasks above must be jointly performed by the players. However, none of them should have "the power of a supervisor;" ideally, the agents should be symmetrical, i.e. should run the same software, possibly driven by different inputs.

For simplicity, we consider a two player setting. For solving point 1, we can assume some policy is defined *a priori* that decides which player has to process a given hook, relying for example on a Type function defined on hooks. The type points to player 1 or to player 2, or to both (in which case both players must process that hook). This defines the necessary exchanges of hooks between players. Concerning point 2, as soon as the blackboard is removed, active hooks stored by the supervisor must be dispatched in the active sets of the players. This raises a difficulty for point 3, since nobody has a complete view of the state of computations. A player can only check that his job is finished locally, but he may eventually receive new hooks to process. Let us define the global end of computations by the following property: all players have finished, and the communication channels are empty. Detecting this stable property is standard in distributed processing, and known as the distributed detection of termination problem. Many algorithms are available in the literature to detect global termination, e.g. Misra algorithm [1, 2]. So we assume this function is incorporated in the players. A distributed VA with two players then takes the following form:

Distributed VA : two players (player i described)

1. initialization $\mathcal{A} := \{h^0\}$, $\mathcal{A}_i := \mathsf{Type}_i(\mathcal{A})$
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
      – $\mathcal{A}_i := \mathsf{Type}_i(\mathcal{A}_i)$
   b. on reception of 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

Vit_i represents steps 2.a,b,c of the centralized VA, where only hooks of type i can be selected for extension. Extending hooks of type i generally produces hooks of both types, whence the sorting in the last two steps of 2.a. Notice that the 3rd step of 2.a only sends newly created hooks, in order no to send twice the same

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

hook to player j. The detection of global termination is stated as soon as local termination is true. The backtrack is more complicated here 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.

Remark. When knowledge of the model is divided between the players, for example by partitioning the transition set, *both* of them must operate on every hook, to ensure that all transitions have been checked for extension of that hook. Hence all hooks are of both types. This is the case below.

2.5 Multisensor distributed VA

We now assume observations are collected by two sensors, located in the players. More precisely, let the transition set T be partitioned into $T = T_1 \cup T_2$, and let us assume that player i only observes transitions of T_i. We thus have two sequences of observations : $o_1 \dots o_{N_1}$ and $o'_1 \dots o'_{N_2}$ (see fig. 1). 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 *one possible interleaving of observations.*

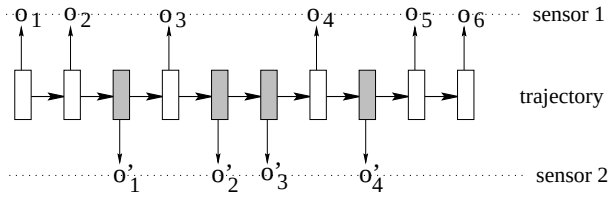

Figure 1: *A trajectory, viewed as a total order of events (represented by blocks). The color determines what set the corresponding transition belongs to. Each set is observed by a dedicated sensor.*

Checking all possible interleavings can be done in a simple way, by replacing index i in hooks by a *double index* (i, j). This indicates a hook compatible with at least one interleaving of $o_1 \dots o_i$ and $o'_1 \dots o'_j$. Extensions augment either i or j, according to the set in which the new transition is taken. In the same way, the reduction selects the most likely hook among those having same state and same double index. This defines a 2-dimensional recursion which contrasts with the 1-dimensional recursion of the standard VA. It represents the overhead one has to pay for checking all possible interleavings. In a distributed systems, the interleaving of concurrent events is meaningless and doesn't have to be recovered. The next section introduces an appropriate concurrency semantics for trajectories, which will significantly decrease this overhead.

3 Distributed VA for subsystems interacting by shared resources

This section extends the previous algorithms to distributed systems, relying on a simple example. Such systems are defined as the composition of elementary subsystems. An appropriate partial order semantics for trajectories is introduced[4, 6, 3], which allows to take true concurrency into account and thus to escape from the interleaving overhead evidenced above. This is not sufficient however, since the algorithms of the previous section handle global states of the system. This notion is not appropriate here, first of all because there is no notion of global time any more, and secondly because the state space is enormous for large concurrent systems. We prove that a distributed algorithm handling local states of subsystems can be designed.

3.1 Systems interacting by shared resources

We model $\mathcal{S}$ as composed of two FSM $\mathcal{S}_A = (A, T_A, L_A, a_0)$ and $\mathcal{S}_B = (B, T_B, L_B, b_0)$. The interaction between $\mathcal{S}_A$ and $\mathcal{S}_B$ is established by another FSM $\mathcal{S}_C = (C, T_C, L_C, c_0)$. 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 and onto mapping $\phi : T_A^s \cup T_B^s \to T_C$. The "distributed 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]). 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 using 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 (see figure 2).

3.2 Partial order semantics for trajectories

For the partial order semantics, trajectories are *partial orders ef events* instead of sequences, so that concurrent events are not ordered in time. They are represented by a *directed acyclic graph (DAG)* of events, usually taken to be the transitive reduction of the partial order (see the notions of *occurence net* in [3], of *puzzle* in [6, 8], or of *unfolding* in [4]). A partial order of events of $\mathcal{S}$ (also called an *event graph*) is a valid trajectory for $\mathcal{S}$ iff its restriction to events of $\mathcal{S}_A$ (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 if the partial order of events coincides with the transitive closure of these three sequences. This definition is illustrated by figure 2, where short blocks correspond to local transitions, long blocks to shared transitions, and where the color identifies transitions of $\mathcal{S}_A$ and $\mathcal{S}_B$. In this picture, the 3rd event of $\mathcal{S}_A$ and the 3rd event of $\mathcal{S}_B$ are concurrent : their ordering is left unspecified.

It is precisely because such orderings are dropped that the overhead of the multisensor distributed VA can be reduced.

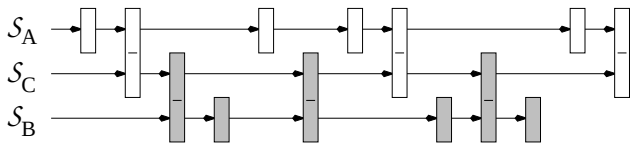

Figure 2: *A trajectory of* $\mathcal{S}_A \| \mathcal{S}_C \| \mathcal{S}_B$.

Partial order semantics are not compatible with the usual definition of stochastic systems, since Markov dynamics can only assign likelihoods to *sequences* of events, whereas we consider partial orders. A special technique called "partial randomization" has been developed to overcome this difficulty [6]. We do not detail it here for lack of space. On the example, it assumes $\mathcal{S}_A$ and $\mathcal{S}_B$ are usual stochastic FSM, and $\mathcal{S}_C$ is not randomized (it somehow reacts to actions of $\mathcal{S}_A$ and $\mathcal{S}_B$). The cost of a trajectory is obtained by summing the costs of the sequences of events on $\mathcal{S}_A$ and $\mathcal{S}_B$.

3.3 Representation of trajectories

Section 3.2 characterizes trajectories of $\mathcal{S}$ in terms of tuples 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 for $\mathcal{S}_A, \mathcal{S}_C$ 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). Extending h with a local event of $\mathcal{S}_A$ (resp. $\mathcal{S}_B$) only changes the h_A (resp. h_B) part, whereas the connection of a shared event also changes the h_C component. With this mechanism, one easily checks that a hook h encodes a unique partial order of events[3]. Observe that such a data structure implicitly contains a double index (i, j), distributed in h_A and h_B. Therefore it is directly amenable to the multisensor framework where a sensor observes transitions of $\mathcal{S}_A$ and another observes $\mathcal{S}_B$.

3.4 Separation property

We establish here the key result that makes possible a distributed algorithm working on local states. 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 h_B represent sequences of events that are respectively compatible with prefixes $o_1 \dots o_i$ and $o'_1 \dots o'_j$ of the observed sequences.

[3]The verification can be done by backtracking from h.

Let us also define the two canonical projections ψ_A and ψ_B by $\psi_A(h) = (h_A, h_C)$ and $\psi_B(h) = (h_C, h_B)$.

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

$$\psi_A(\mathcal{A}) \otimes \psi_B(\mathcal{A}) \triangleq \{(h_A, h_C, h'_B) \,/\, (h_A, h_C) \in \psi_A(\mathcal{A}), \; (h'_C, h'_B) \in \psi_B(\mathcal{A}), \; h_C = h'_C\}$$

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 figure 2). In other words, given that they agree on the interleaving of shared transitions, trajectories in $\mathcal{S}_A$ and $\mathcal{S}_B$ can vary freely: the "holes" between consecutive shared transitions can be filled by any (legal) sequence of local transitions.

3.5 Fully distributed algorithm

The multisensor distributed VA is readily applicable to hooks h with the special extensions we have defined. This allows to implement the partial order semantics that associates the same cost to different interleavings of concurrent events. However, by working on global states, the algorithm does explore all interleavings, which is useless now. And it also suffers from the high cardinality of the state space. To overcome these drawbacks, we develop a new version working on *local states* of subsystems. Relying on proposition 1, we represent the set of active hooks under a *product form*, i.e., through the one-to-one correspondence $\mathcal{A} \leftrightarrow \psi_A(\mathcal{A}) \otimes \psi_B(\mathcal{A})$. Naturally, "player A" observing transitions of $\mathcal{S}_A$ will be in charge of the $\psi_A(\mathcal{A})$ part, denoted by $\mathcal{A}_A$, and symmetrically "player B," observing $\mathcal{S}_B$, will handle the $\psi_B(\mathcal{A})$ part, denoted by $\mathcal{A}_B$. Primitive operators need to be modified however.

Extension. Both players must work on each active hook h. So when player A creates a new hook, it must communicate it to player B. In the distributed state setting, this means that each time player A changes the h_C component into h'_C, it must communicate this change to player B. We thus keep track of these changes in the extensions, whence the notation: $(\mathcal{N}_A, \mathcal{C}_A) := \mathsf{Ext}_A(\mathcal{A}_A)$, where $\mathcal{N}_A$ are the new pairs (h'_A, h'_C), and $\mathcal{C}_A$ stores the changes $h_C \to h'_C$.

Update. This operator is activated on reception of a message $\mathcal{C}$ to incorporate the actions of the other player on the shared system. For player A, it is defined by

$$\mathsf{Update}(\mathcal{A}_A) = \{ \, (h_A, h'_C) \,/\, (h_A, h_C) \in \mathcal{A}_A, \; (h_C \to h'_C) \in \mathcal{C}_B\}$$

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.

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

That is, the selection on histories of $\mathcal{S}_A$ is conditioned by the fact that they finish in the same state *a and* that they are synchronized with the same *history* h_C on $\mathcal{S}_C$. Which means that the actual notion of "state" for recursive optimisation in the local Viterbi is the pair $(X(h_A), h_C)$, and *not* $(X(h_A), X(h_C))$.

Fully distributed algorithm : (player A described)

1. initialization $\quad \mathcal{A}_A := \{(h_A^0, h_C^0)\}, \quad \mathcal{W}_A := \emptyset$
2. forward sweep : until `global-end` is detected
   a. on decision of processing hooks
      - select $\mathcal{A}'_A \subset \mathcal{A}_A$
      - $(\mathcal{N}_A, \mathcal{C}_A) := \mathsf{Ext}_A(\mathcal{A}'_A)$
      - $\mathcal{A}''_A := \mathsf{Red}_A(\mathcal{N}_A)$
      - $\mathcal{A}_A := (\mathcal{A}_A \setminus \mathcal{A}'_A) \oplus \mathcal{A}''_A$
      - $\mathcal{W}_A := \mathcal{W}_A \oplus \mathcal{A}'_A$
      - send $\mathcal{C}_A$ to player B

   b. on reception of message $\mathcal{C}_B$ from player B
      - $\mathcal{A}_A := \mathcal{A}_A \oplus \mathsf{Update}(\mathcal{A}_A, \mathcal{C}_B) \oplus \mathsf{Update}(\mathcal{W}_A, \mathcal{C}_B)$

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

By contrast with previous algorithms, hooks that have been processed are stored in a "waiting set" $\mathcal{W}_A$, since they can be modifed by the actions of the other player on their h_C component, in which case they are reactivated. To initiate the backtrack, the most likely h must be determined, by summing local costs of pairs (h_A, h_C) and (h'_C, h'_B) that coincide on the h_C part. This can be done with a simple protocol.

4 Conclusion

We have presented a distributed state reconstruction algorithm for a discrete event system defined as the parallel composition of subsystems. The structure of the algorithm parallels the structure of the system : it is based on asynchronous agents, one per subsystem, with local knowledge of the model and of observations. This architecture is made possible by the partial order semantics on trajectories, which exploits explicitly the concurrency of events. It is particularly suited to large concurrent systems, and offers an alternative to the curse of dimensionality since only local states are handled. The extension to N subsystems is being investigated, which allows to design Bayesian Networks of dynamic systems. An application to the monitoring of telecommunication networks is being developed, 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.