FAULT DIAGNOSIS FOR DISTRIBUTED ASYNCHRONOUS DYNAMICALLY RECONFIGURED DISCRETE EVENT SYSTEMS.

S. Haar, A. Benveniste, E. Fabre, and C. Jard

Abstract: Diagnosis of concurrent and asynchronous systems, such as large telecommunication or information systems, requires powerful mathematical models. The use of Petri net unfoldings allows to formalize diagnosis using partial order semantics, a generalization from the global state model imposed by the use of automata. If, in addition to asynchronicity and distribution, the network topology itself is subject to dynamic changes, all static models, including Petri nets, reach their limits. Then, graph grammars can be used, encoding in the current local states not only the current values of state variables but also the current topology of the network connections; the fact that unfolding semantics is available allows to carry over the diagnosis algorithms to this setting.

Keywords: Networks, Discrete event systems, Fault Diagnosis, Distributed models

1. INTRODUCTION

Diagnosis of concurrent and asynchronous discrete events dynamical systems, such as large telecommunication or information systems, is a challenging topic, see Benveniste et al. (2003). Suppose a system is monitored in such a way that at each site, alarms are collected at a single sensor and processed by a central supervisor. Since communications are asynchronous, each local sensor has only a partial view of the system, and its local time is not synchronized with that of the other sensors. Time is totally ordered at each individual node. But it is only partially ordered between different nodes. Even if the order of events may be correctly observed by each individual sensor, communicating alarm events via the network causes a loss of synchronization. Hence the interleaving of events communicated to the supervisors is nondeterministic. Global states of the overall system are very costly to capture, and are in fact never handled in a large networked system. Thus we use only local states attached to each individual node.

⋆ Supported by the RNRT (French Research Ministry) projects MAGDA2 and SWAN. Address: IRISA, Campus de Beaulieu, 35042 Rennes cedex, France. A.B., E.F., S.H., are with Inria, C.J. with CNRS. Corresp. author: Stefan.Haar@irisa.fr

We will sketch here the approach of Benveniste et al. (2003) to represent all solutions of the (centralized) diagnosis of concurrent and asynchronous systems, by using so-called unfoldings, see Engelfriet (1991); we will illustrate the techniques below in the context of a toy client-server model.

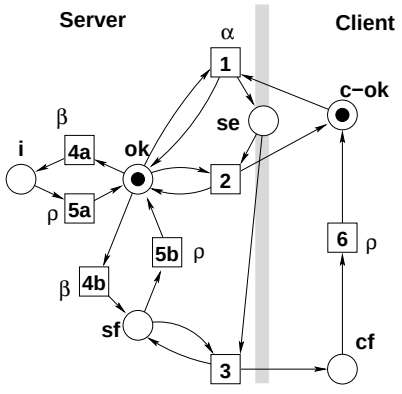

Fig. 1. Running Petri net example

In *dynamic* settings, this approach reaches its limits: **(i)** in *network management*, insertion of new peers modifies the connectivity structure and requires dynamic and local (re-)negotiation of services changing the structure to be supervised, and/or the set of su-

pervisors. Dually, withdrawal of peers, loss of connections etc, reduce the connectivity graph. **(ii)** *Web services* requested by a client peer can be performed by a server peer itself or via secondary service calls to other peers, etc; the actual topology of interaction will change depending on availability, parameters. etc. We will illustrate the use of *Graph grammars* as a unifying formal model that supports diagnosis and improves over Petri nets by allowing self-reconfiguration; the running example will be modified into a dynamic one-server system with several on-and-off clients.

The paper is organized as follows: we illustrate in Section 2 the diagnosis with unfoldings, in the context of Petri nets. Section 3 introduces graph grammars as a model for dynamicity, and their unfoldings. Section 4 discusses an example of one server with two clients, extending the running example from the Petri net case, and Section 5 concludes.

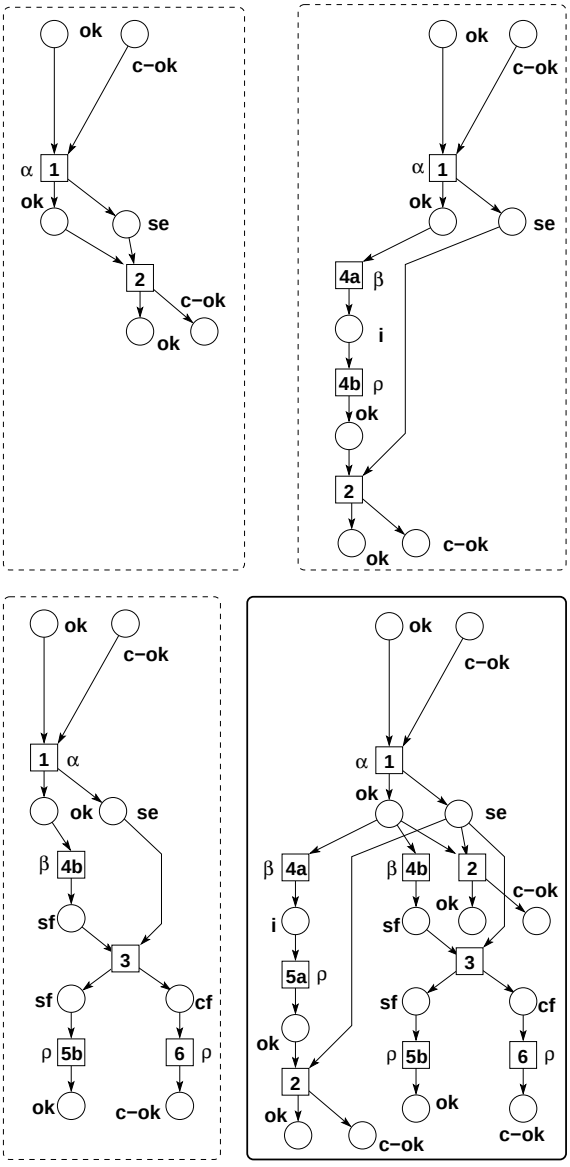

Fig. 2. Unfolding of the running example

2. STATIC TOPOLOGY : PETRI NET APPROACH

Suppose first a fixed network topology, and that all variables involved have finite domain; we can then use safe Petri nets with so-called "true concurrency semantics" to formalize distributed fault diagnosis. We assume the reader is familiar with terminology and representation of Petri nets.

Our example is shown in Fig. 1. It has two interacting components, server and client. The client may fail due to server failure but not vice versa. In our model, the client is either *ok* (token in the place **c-ok**), being served (**se**), or in a fault state (**cf**). The server can be **ok**, in an internal fault state **i** or in a visible fault **sf** that may effect the client. The transitions of Figure 1 are: server enters (**4a**) or leaves (**5a**) internal fault state; same for visible fault (**4b**, **5b**); service begins (**1**) and may end normally with client "ok" (**2**), or abnormally due to server fault, with the client in the fault state, with recovery (**6**). Note that an internal server fault can occur concurrently with ongoing service; it only delays the end of service, while visible fault can prevent normal termination.

Suppose now that the server emits alarm β upon entering one of its fault states; that is, we label transitions **4a** and **4b** by β. In the same way, label the repair transitions **5a**, **5b** and **6** by ρ, and service beginning **1** by α; the unlabeled transitions represent unobservable events. - The initial marking is given by the set $\{\mathbf{av}, \mathbf{ok}, \mathbf{id}\}$. Labels ($\alpha, \beta$ or self-repair ρ) attached to the different transitions or events, will be called *alarms* in the sequel. The **diagnosis problem** is now to identify, for a given sequence A of alarms (we only consider sequential observations here to keep the presentation simple), all possible behaviours of the system (represented by its model, which we assume complete for the moment) that would have triggered A, i.e. represent possible *explanations* for A.

Unfoldings represent all runs. he partial order semantics known as *branching processes* gives nonsequential executions of the net in a compact form. Figure 2 shows three different scenarios, and the same with their common prefixes glued together (bottom right): the latter structure is a branching process, a prefix of the unfolding of the net. Its graph forms an *occurrence net*, see below; to distinguish from the corresponding concepts in Petri nets, we speak of *conditions/events* instead of places/transitions. The construction is recursive: let Petri net $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \rightarrow, M_0)$ be given. For each place in the initial marking M_0, create a copy, an initial *condition*. Then, add recursively **1)** a new event e for each transition t that is enabled in the corresponding marking, with arcs to e from the condition set (*co-set*, see below) corresponding to t's pre-places, and **2)** output arcs leading to new output conditions for e , one for each place that receives a token from t; repeat. The unique maximal element of the set of branching processes thus constructed is called the *unfolding* $\mathcal{U}_\mathcal{N}$ of Petri net

$\mathcal{N}$, see Engelfriet (1991). Conditions/events $\mathcal{U}_\mathcal{P}$ are labeled by places/transitions of $\mathcal{P}$ such that input and output sets of transitions are respectively in bijection. Consider Figure 2. The first scenario shows a normal service cycle, where the server is "ok" consistently. In the second, the service starts and ends normally but its termination is delayed by an internal server fault; in the third, service is disrupted following a visible server fault. The last figure (bottom right) is the branching process formed by these scenarios, identifying their shared prefixes. Note that all scenarios and the branching process are acyclic by construction; the transitive closure of the constructed flow arc relation is a partial order $\leqslant$, called the *causality* relation. The preset of any condition contains *exactly* one event (or none if the condition is initial), and *in each scenario*, the postset of any condition contains *at most* one event. In a branching process, a condition's postset can contain two or more different events, as shown by the first condition labeled $\mathbf{se}$ on the right hand side, whose post-events are occurrences of 2 (twice) and 3. This indicates a *conflict*, corresponding to the fact that in each run, at most one of those events can occur. Conflict is *hereditary*: if $x\#x'$ and $x \preceq y, x' \preceq y'$, then $y\#y'$ follows. Finally, *concurrency* holds between nodes x and y that are neither in conflict (thus there exists a run containing both) nor causally ordered, i.e. neither $x\#y$, nor $x \preceq y$, nor $y \preceq x$: this is the case for $5a$ and 6 in the third scenario. In particular, the initial conditions form a maximal set of pairwise concurrent conditions; such sets are called *cuts*. Each cut corresponds to a reachable marking, and each reachable marking is reflected by at least one cut of the unfolding. Each branching process of a Petri net $\mathcal{N}$ associates to $\mathcal{N}$ an *occurrence nets (ON)*, which we define now: $ON = (\mathcal{B}, \mathcal{E}, \rightsquigarrow, \mathbf{c}_0)$ is an ON iff

(1) $\rightsquigarrow$ is acyclic, and defines a partial order $\leqslant$ without infinite descending chains;
(2) no node is in conflict with itself : $\neg(y\#y)$;
(3) the set $\mathbf{c}_0$ of $\leqslant$-minimal nodes is a cut, called the *initial cut* of ON.

The executions of a Petri net are reflected in suboccurrence nets of its unfolding. Call *configuration* every subnet κ of ON that **(i)** contains the initial cut $\mathbf{c}_0$, **(ii)** is conflict-free, i.e. does not contain any conflicting pair of nodes, and **(iii)** is causally closed, i.e. $x \in \kappa$ and $y \leqslant x$ together imply $y \in \kappa$. Maximal configurations are called *runs*.

Asynchronous diagnosis. In Fig. 2, assume we are given alarm sequence sequence $A = \alpha\beta\rho$ recorded at the unique sensor. The configuration κ_1 top left produces neither β nor ρ, so is not an explanation for A. Top right, κ_2, fits A, we include it in the diagnosis of A. The third, κ_3, produces an alarm pattern containing A as a prefix: it has two prefixes that explain A (the ones containing exactly one ρ-labeled event) See Benveniste et al. (2003) for the formal algorithm, using synchronized Petri net products.

3. DYNAMIC NETWORKS AND THE GRAPH GRAMMAR MODEL

Once the assumption of fixed network topology is dropped, the Petri net model is no longer applicable; however, concurrency remains present. We will see that true concurrency semantics carries over and allows to extend the above asynchronous diagnosis approach, provided *graph grammars* are used as system model. The principle of rewriting using rules in the case of transformation systems that operate on graphs is the same as that for generators of word languages; however, this is not our concern here. We view graph grammars as *discrete event dynamical systems*, generated by an initial graph and a collection of rules; they may exhibit concurrency in the parallel application of rules. Several classes of graph grammars can be found in the literature (see references); we mention node rewriting systems, hyperedge replacement systems, SPO grammars. Our presentation focuses on the class of **Double Pushout (DPO) Grammars**. To fix terminology, let $\mathcal{V}$ be a non-empty set of *vertices*, $\mathcal{E}$ a set of *(hyper)edges* (we will drop henceforth the prefix "hyper": whenever we speak of "edges" and "graphs", we mean "hyperedges" and "hypergraphs"), and sc, tg two mappings associating to each edge $\mathsf{e} \in \mathcal{E}$ sets $sc(\mathsf{e}), tg(\mathsf{e}) \subseteq \mathcal{V}$ of source and target vertices, respectively. Note that $sc(\mathsf{e})$ and $tg(\mathsf{e})$ need not be disjoint, and that either set can be empty. $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is called a **(hyper)graph**. A **graph morphism** $\varphi : \mathcal{G}_1 \to \mathcal{G}_2$ is a pair of partial mappings $\varphi_\mathcal{V} : \mathcal{V}_1 \to \mathcal{V}_2$ and $\varphi_\mathcal{E} : \mathcal{E}_1 \to \mathcal{E}_2$ such that for all $\mathsf{e}_1 \in \mathcal{E}_1$ such that $\varphi_\mathcal{E}(\mathsf{e}_1)$ is defined, $\varphi_\mathcal{V}$ induces bijections $sc_1(\mathsf{e}_1) \leftrightarrow sc_2(\varphi_\mathcal{E}(\mathsf{e}_1))$ and $tg_1(\mathsf{e}_1) \leftrightarrow tg_2(\varphi_\mathcal{E}(\mathsf{e}_1))$.

(Hyper-)Graph transformation. Consider Figure 3. It shows a replacement of a hyperedge labeled a by a hypergraph of the form given by the graph "R". Here, the grammar rule is given by the upper line, $L \leftarrow K \rightarrow R$, relating an abstract left hand side graph L to the right hand side graph R that replaces L, up to preservation of the shared part of the form given by the interface graph K. Formally, the interface K is mapped by injective morphisms (arrows) into both L and R. Then, an occurrence of L in G is detected: that is, there exists another injective morphism inserting L into G, called *matching* (it is probably more intuitive to view this matching as the *detection* of a subgraph

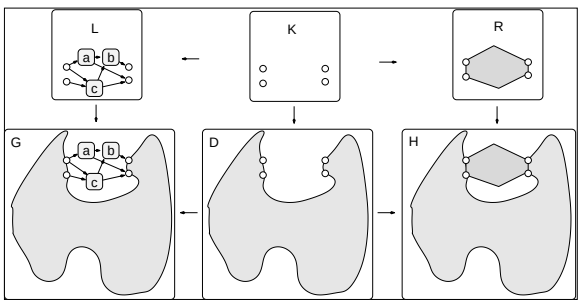

Fig. 3. The elements of a graph transformation rule

of G that is isomorphic to L). The categorical **double pushout (DPO)** construct (see Ehrig et al. (1999); Baldan et al. (1999)) fills in the remaining graphs and morphisms such that: the diagram's squares commute, the abstract interface K is matched in D, and D is matched in both G and the resulting graph H, and Graph H is obtained as the "pushout" of $D \leftarrow K \rightarrow R$: that is, the fusion of an image of R and D along the common interface of form K. D is to be interpreted as the context graph that remains unchanged under the production; Figure 3 illustrates the application of a rule to graph G, producing H.

Definition 1. A **typed graph** over graph $\mathcal{TG}$ is a pair $(\mathcal{G}, typ)$ where $\mathcal{G}$ is a graph and $typ : \mathcal{G} \rightarrow \mathcal{TG}$ a morphism assigning to each node n and edge e in $\mathcal{G}$ a corresponding node / edge in $\mathcal{TG}$, called the **type**. A **typed DPO graph grammar** is a tuple $\mathbf{G} = (\mathcal{TG}, \mathcal{G}, typ, \mathbf{R})$, where $\mathcal{TG}$ is a type graph, $(\mathcal{G}, typ)$ a typed graph over $\mathcal{TG}$, and $\mathbf{R}$ a set of DPO productions $\mathbf{r} : (L \xleftarrow{l_\mathbf{r}} K \xrightarrow{r_\mathbf{r}} R)$ with $l_\mathbf{r}, r_\mathbf{r}$ injective graph morphisms.

For technical reasons, it is preferable to restrict attention to rules that preserve vertices (see Baldan et al. (1999) for a discussion): that is, we assume that for all rules $\mathbf{r} : (L \xleftarrow{l_\mathbf{r}} K \xrightarrow{r_\mathbf{r}} R)$ and vertices v in L, v is in the image of K under $l_\mathbf{r}$ (this implies that v is preserved by $\mathbf{r}$, i.e. has a counterpart in R).

Occurrence Grammars and Configurations. Rules exhibit relations similar to events in occurrence nets:

Definition 2. Let $G \leftarrow D \rightarrow H$ be an instance of the rule $\mathbf{r} = (L \leftarrow K \rightarrow R)$. Denote as

(1) ${}^\bullet\mathbf{r}$ the set of type graph elements **consumed** by $\mathbf{r}$
(2) $\mathbf{r}^\bullet$ the set of elements **produced** by $\mathbf{r}$, and as
(3) $\underline{\mathbf{r}}$ the set of elements **preserved** by $\mathbf{r}$

of a grammar $\mathbf{G}$ is defined as follows: Let $\mathbf{r}_1, \mathbf{r}_2 \in \mathbf{R}$, and x a node or edge in $\mathcal{TG}$. The **causal relation** $<$ is defined as follows:

(1) If $x \in {}^\bullet\mathbf{r}_1$ then $x < \mathbf{r}_1$;
(2) if $x \in \mathbf{r}_1{}^\bullet$ then $\mathbf{r}_1 < x$;
(3) if $\mathbf{r}_1{}^\bullet \cap \underline{\mathbf{r}_2} = \emptyset$ then $\mathbf{r}_1 < \mathbf{r}_2$.

The **asymmetric conflict relation** $\nearrow$ on $\mathbf{R}$ is:

(1) if $\underline{\mathbf{r}_1} \cap {}^\bullet\mathbf{r}_2 \neq \emptyset$ then $\mathbf{r}_1 \nearrow \mathbf{r}_2$;
(2) if ${}^\bullet\mathbf{r}_1 \cap {}^\bullet\mathbf{r}_2 \neq \emptyset$ and $\mathbf{r}_1 \neq \mathbf{r}_2$ then $\mathbf{r}_1 \nearrow \mathbf{r}_2$;
(3) if $\mathbf{r}_1 < \mathbf{r}_2$ then $\mathbf{r}_1 \nearrow \mathbf{r}_2$.

To illustrate Definition 2, consider Figure 4. Rule $\mathbf{r}_0$ may occur before $\mathbf{r}_2$, but not after $\mathbf{r}_2$; on the other hand, there is no causal link between the two, since $\mathbf{r}_2$ does not require $\mathbf{r}_0$. We thus have $\mathbf{r}_0 \nearrow \mathbf{r}_2$, which can be read as "$\mathbf{r}_0$ is **prevented** by $\mathbf{r}_2$". Similarly, $\mathbf{r}_0 \nearrow \mathbf{r}_2$; there are three possible executions, with $\mathbf{r}_2$ alone, or e_2 after $\mathbf{r}_0$, or $\mathbf{r}_2$ after $\mathbf{r}_1$; there is no possible execution with *both* $\mathbf{r}_0$ and $\mathbf{r}_1$, which is consistent with the fact that $\mathbf{r}_0 \# \mathbf{r}_1$. A binary conflict relation "$\#$" containing and extending the above notion of conflict in occurrence nets is obtained as the symmetric closure of $\nearrow$, hence $x \# y$ implies $x \nearrow y$ but not

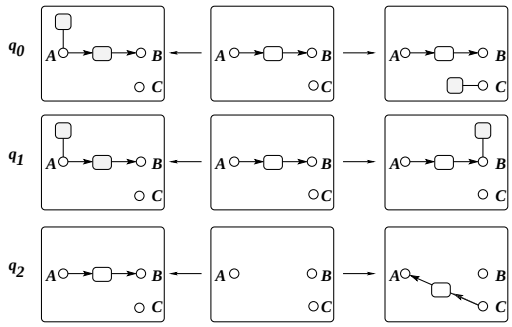

Fig. 4. *Asymmetric Conflict: Rule $\mathbf{r}_0$ is prevented by $\mathbf{r}_2$, and in conflict with rule $\mathbf{r}_1$*

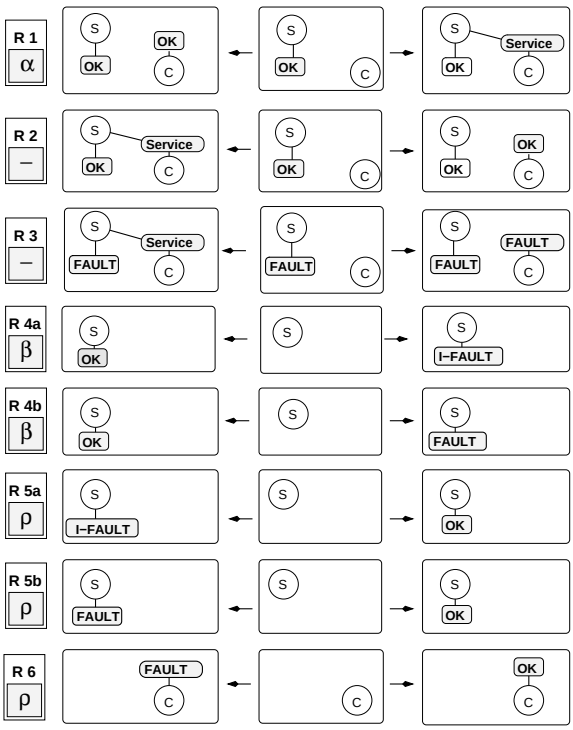

Fig. 5. *Rules of a server-client system with associated alarm labels*

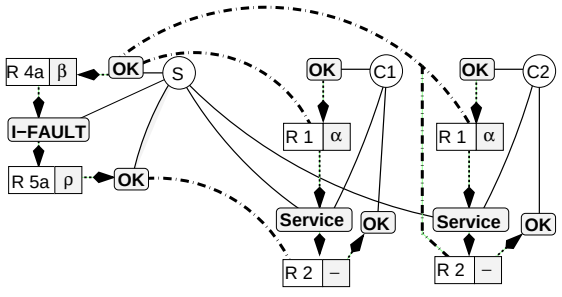

Fig. 6. *One service with regular behaviour and a concurrent internal fault that delays termination*

vice versa.We note that in the presence of asymmetric conflicts, any event e can have several histories, whose difference lies in the occurrence (or not) of events that are prevented by e or an ancestor of e but which do not interfere with e itself.

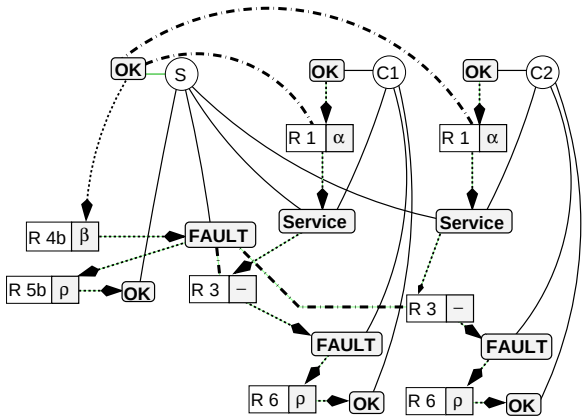

Fig. 7. *Visible fault disrupting all services*

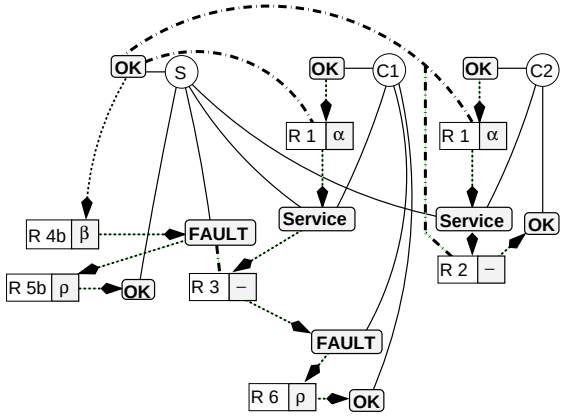

Fig. 8. *Visible fault disrupting one service while allowing another to terminate regularly*

4. EXAMPLE: MULTI-CLIENT-SYSTEM

Let us now consider the above example which we modify by adding dynamicity : suppose one server may be serving an unspecified number of clients; for simplicity, we restrict to a setting with two clients present, so two, one, or none may be receiving service at a given instant. Then the servers state changes from ok to $faulty$ will also affect a varying number of clients, namely, those that are currently being served. The Petri net model above had server and client invariably linked. The dynamic multi-client case cannot be handled in the same way (more precisely, one could model each service configuration by alternative parts in a bigger Petri net: the readers may want to perform this operation for the small example here, and convince themselves of the exponential growth of the model); we need a more supple formal representation. Figure 5 shows the rules of a graph grammar model for this situation; the rule names and alarm labels are indicated. Note that in the figures showing graph transformations, we draw circles to represent vertices; rectangular boxes with rounded corners represent hyperedges, and those with unrounded corners, occurrences of rules. So, in the left hand side of $R1$, generic server S is represented by a vertex; the two hyperedges having S as unique source and target vertex (call

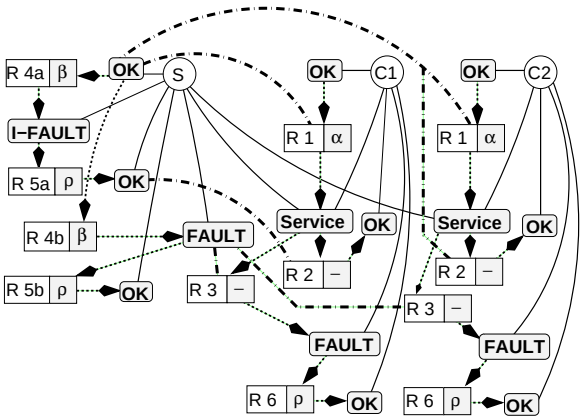

Fig. 9. *Unfolding prefix containing all of the above configurations, with alarm labels*

this type *1-edges*) represent the "ok" state. Similarly, a generic client C is a vertex : we adopt the principle of representing actors by vertices, states by 1-edges, and connections, multirelations etc. by more general hyperarcs reflecting the topology of interaction. By the action of $R1$, a service is established between server S and generic client C; C's "ok" arc disappears, and an arc connects S and C while the service is being delivered. (One has bigger edges in other examples : suppose for instance that C's request concerns a certain number of other nodes, e. g. objects to be repaired, contacts with other clients to be brokered, etc; one would then link all the vertices concerned by one hyperedge, ensuring that future transactions are applied consistently to the correct "case file" in all sites.) Note that the "ok" status of the server must hold during the entire transaction; it is therefore part of the interface graph preserved by $R1$. The regular termination of service is described by $R2$: the service-arc is removed, and the client returns into "ok". In $R3$, the effect of a visible server fault is shown: the arc for service en route is removed, and C enters a fault state. In rules $R4a$ and $R4b$, S leaves the "ok" state and enters one of two fault states : internal (IFault, $R4a$) or visible (Fault, $R4b$), with external effects given by $R3$. Rules $R5a$ and $R5b$ describe repair of faults of S. The only rule that concerns the generic client C alone is $R6$, where C repairs its fault (such as induced by a server fault during service) and returns into the "ok" state. Note that this behaviour reflects the one given by the Petri net example above; in fact, we have here a generalization of the 1-Client/1-Server system to several clients that may join or leave. In fact, the number of clients or servers is not specified by the rule set; it will be given by the initial graph of a process. The rule set can also be extended by "birth" rules creating servers or clients from nothing, and / or "death" rules in which an idle client looses that edge, and so forth. The figures here show only a space-restricted selection of possibilities.

Unfolding. The Figures 6 through 8 show configurations of the above example. Circles are vertices, round boxes with solid lines represent hyperedges. The rect-

angular boxes represent the occurrence of the rule whose names and labels are inscribed; consumption and creation of graph parts are indicated by dotted *arrows*, and thick dash-dotted *lines* indicate side conditions. Side conditions are indicated by *read arcs* (dash-dotted lines) In all cases, there is initially one server in the "ok" state, and two clients $C1$, $C2$ initially "ok". All clients are instances of the generic client C in the grammar rules from Figure 5. In Figure 6, $C1$'s request is entirely satisfied: $R1$ starts services, linking $C1$ and $C2$ to S; service for $C2$ terminates without difficulties. During the service for $C1$, but *after* successful termination for $C2$, an internal fault occurs at S ($R4a$) and is repaired ($R5a$); only after repair can the service terminate normally ($R2$), and $C1$ then returns into the idle state. Note that the side conditions of $R1$ and $R2$ for $C1$ force this ordering of $R4a$ and $R2$: $R2$ acts only after repair, not during the first "ok" state - the side condition of $R1$'s occurrence - since in that case we would have had a read arc from that first "ok" to *both* $R1$ and $R2$.

In Figure 7, both $C1$ and $C2$'s services are prevented from normal termination by the visible fault of S and produce faults; after repairs, S and the clients turn "ok", but no service was achieved. In Figure 8, the service for $C2$ terminates normally and sufficiently fast to escape the visible fault on S, which disrupts the service for $C1$, with the usual repair afterwards. Figure 9 gives a comprehensive view of all the above executions; it shows a prefix of the grammar's unfolding, with shared prefixes glued together like in the Petri net unfolding of Figure 2.

Histories and asymmetric conflicts. Let us take another close look at Figure 7. One notices the presence of asymmetric conflict: the two occurrences of $R1$ are prevented by $R4b$, and similarly, those of $R3$ by $R5b$. The occurrence of $R5b$, call it e, has several different causally closed histories within the configuration: the event set $\mathcal{H}_1(e) = \{R1_1, R1_2, R3_1, R3_2, R4b\}$, where we note Rx_y the occurrence of rule x at client y [1], and the event sets

$\mathcal{H}_2(e) = \{R1_1, R1_2, R3_1, R4b\}$, $\mathcal{H}_3(e) = \{R4b\}$
$\mathcal{H}_4(e) = \{R1_1, R1_2, R3_2, R4b\}$,
$\mathcal{H}_5(e) = \{R1_1, R1_2, R4b\}$, $\mathcal{H}_6(e) = \{R1_1, R4b\}$
$\mathcal{H}_7(e) = \{R1_1, R3_1, R4b\}$, $\mathcal{H}_8(e) = \{R1_2, R4b\}$
$\mathcal{H}_9(e) = \{R1_2, R3_2, R4b\}$.

These sets are in fact mutually exclusive, potential histories explaining e; the asymmetric conflict prevents, e.g., $R1_2$ to occur after $\mathcal{H}_4(e)$, or $R3_1$ after $\mathcal{H}5(e)$. The presence of e allows neither to deduce nor to exclude occurence of $R1$ and $R3$.

Diagnosis. Now, using *labels* one can perform fault diagnosis procedure as for Petri net unfoldings : according to Figure 5 all self-repair rules are labeled ρ, server faults as β, and α is the label of applications of $R1$. The label sequence $A = \alpha\alpha\beta\rho\rho\rho$ is explained by the configuration of Figure 7, but not the other two: Figure 6 allows only $\alpha\alpha\beta\rho$, and Figure 8 only $\alpha\alpha\beta\rho\rho$. Thus sequence A filters away those configurations, as not being candidates for diagnosis. The formal technique for diagnosis over graph grammars can be obtained by lifting from the Petri net approach sketched above:

(1) describe a given alarm pattern A as a deterministic (conflict-free) graph grammar $\mathbf{G}_A$;
(2) unfold the product grammar $\mathbf{G}_A \times \mathbf{G}$ obtained from $\mathbf{G}_A$ and the system model $\mathbf{G}$ by synchronizing rules bearing the same label ;
(3) take the set $\mathbf{diag}'(A)$ formed by all those configurations κ' in the unfolding of $\mathbf{G}_A \times \mathbf{G}$ such that the projection of κ' to the label set yields A (and not a proper prefix); then,
(4) diagnosis of A is the set $\mathbf{diag}(A)$ of projections of all $\kappa' \in \mathbf{diag}'(A)$ to rules from $\mathbf{G}$.

5. CONCLUSION

Asynchronous dynamic systems whose topology change dynamically can be modelled and diagnosed by graph grammars using algorithms that extend naturally those developped for Petri nets. The presence of interfaces introduces asymmetric conflict, leading to slightly "blurred" causal histories, but which can still be discriminated using adaquate labeling for diagnosis. It should be noted that asymmetric conflict is common to systems with **read arcs**, such as *Read Petri Nets*, see Baldan et al. (2001).

[1] note that the occurrences of $R6$ in Figure 7 are concurrent with e, not part of its causal history

REFERENCES

Baldan, P.; Corradini, A; and Montanari, U. Unfolding and event structure semantics for graph grammars. *Proc. FOSSACS 1999, LNCS* **1578**:73–89.

Baldan, P.; Corradini, A; Montanari, U. Contextual Petri nets, asymmetric event structures and processes. *Information and Computation* 171(1):1–49, 2001.

Benveniste, A; Fabre, E.; Haar, S; and Jard, C. Diagnosis of asynchronous discrete event systems, a net unfolding approach. *IEEE Transactions on Automatic Control*, 48(5):714–727, May 2003.

Ehrig, H; Kreowski, H.-J.; Montanari, U; Rozenberg, G. (eds.). *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. 3.* World Scientific 1999.

Engelfriet, J. Branching processes of Petri Nets. *Acta Informatica*, 28:575–591, 1991.

Benveniste, A; Fabre, E.; Haar, S; and Jard, C. Distributed monitoring of concurrent and asynchronous systems. In *Proc. CONCUR 2003 (Invited Talk)*, LNCS. Springer, 2003. Revised and extended version to appear in *Discrete Event Systems*.