Partitioning of Hierarchical Automation Systems

Emanuele Ciapessoni\textsuperscript{1} Stefano Crespi-Regghizzi\textsuperscript{2} Francesco Maestri\textsuperscript{1}
Andrea Ornstein\textsuperscript{2} Giuseppe Psaila\textsuperscript{3} Judit Szanto\textsuperscript{1}

\textsuperscript{1}CESI Centro Elettrotecnico Sperimentale Italiano Giacinto Motta s.p.a.
Via Rubattino 54 - I-20134 Milano (MI), Italy
e-mail: ciapessoni.emanuele@cesi.it, maestri@cesi.it, szanto@pea.enel.it
\textsuperscript{2}Politecnico di Milano - Dipartimento di Elettronica e Informazione
Piazza L. Da Vinci, 32 - I-20133 Milano, Italy
e-mail: crespi@elet.polimi.it, aornstein@iol.it
\textsuperscript{3}Universit`a degli Studi di Bergamo - Facolt`a di Ingegneria
Viale Marcon 5 - I-24044 Dalmine (BG), Italy
e-mail: psaila@unibg.it

Abstract

The research described concerns the partitioning of large control applications for a multi-computer system in order to meet plant localization requirements and to exploit parallelism. The considered applications have hierarchical structure and are composed by a network of automata.

Our application domain is the automation of power stations and electricity distribution. Because of strong EM noise in such environments, the software architecture is organized to be tolerant to transient faults, which could affect the stability of the control system. The hierarchical structure provides a decompositional approach to the design of complex applications. The context for this work is the ASFA platform, originally designed by the Italian board of electricity. The main result is a new partitioning algorithm for hierarchical automata networks, that splits the application into subnetworks which are deadlock-free, compliant with localization constraints, and as parallelizable as possible. The algorithm is also able to satisfy mutual exclusion constraints and to take into account computation/communication weights to achieve balancing of partitions.

1 Introduction

This paper describes recent research on the hierarchical, transient-fault-tolerant approach for power automation systems and electricity distribution systems, allowing a more effective use of multi-computer architectures. In order to introduce the goals and results we have to outline and motivate the special software architecture of such control systems.

Plant automation systems for electric power stations and electricity distribution systems are complex, real-time programs operating in a hostile environment subject to strong EM fields, that may cause transient faults. The prevalence of transient and to some extent predictably periodical faults motivated the choice of different solutions for such systems with respect to the classical fault-tolerance methods of other industrial sectors. The essential idea is to model the system as a huge, cyclic state machine with state variables that must be preserved in stable memory for a duration longer than the expected EM disturbance, so that at the next input cycle computation can resume from a valid state. This ideal model would not work in practice because complex plants require many thousand states, too many for any software designer. Therefore the automation system has to be decomposed into a network of communicating automata, a solution not uncommon in other real-time control systems specified using StateChart or UML state-modeling features. But a generic network of asynchronous state machines would miss the basic requirements of transient-fault tolerance. In order to match both requirements of compositionality and fault-tolerance, ENEL (the Italian board of electricity) has introduced long ago the ASFA software architecture [3, 1], which has been successfully applied to large plants; the ownership of the ASFA software is now of CESI, a company of the ENEL Group which conducts advanced developments concerning the field of electricity production and distribution.

The essential idea is to impose a rigid hierarchy on the network topology, which is organized as a tree. The automata on the leaves collect and issue signals to the plant. The automata in higher positions of the tree implement decision functions, in a bottom-up sweep. The global state of the network is thus the product of the individual states, which can be made tolerant to transient faults. The results of the decision process has to be transformed into actuator signals and output to the plant. This is performed by a symmetrical top-down sweep. This two-sweeps hierarchical architecture is the object of the reported development, that is
The paper is organized as follows. In Section 2 we recall and formalize the ASFA two-sweep hierarchical architecture using graph-theoretical concepts. In Section 3 the partitioning problem is described and formalized. In Section 4 the partitioning algorithm is presented and justified. In Section 5 the effectiveness of the algorithm is experimentally evaluated on a real case study. Finally, Section 6 concludes.

2 ASFA

ASFA is a design and specification technique for hierarchical automation systems. It is based on the same principle on which every control system is based: the control system can be viewed as a black box which receives a set of input signals from the plant and generates commands to the plant through output signals.

As in other proposals, in an ASFA control system the basic processing unit is an automaton; thus the overall control system is a collection of cooperating automata.

Industrial control systems based on finite-state machines (see [5]) can give rise to very complex automata systems, whose definition is difficult, due to the large number of automata and of interconnections among them. Modularity constructs and abstractions should be applied, but often they are lacking or insufficient.

ASFA extends the classical automata-based model for control systems introducing concepts which make possible the orderly modular design of the automaton network at different levels of abstraction.

2.1 Automaton Hierarchy

The main idea provided by ASFA is the so-called automaton hierarchy. We said that an ASFA system is constituted by a set of automata. In order to dominate the growth of the complexity when the number of automata becomes significant, an ASFA network based on a hierarchical structure is built. In practice, a pool of automata directly monitor and process the signals coming from the plant, filter them and in turn send signals/communications to other automata, placed at a superior level w.r.t. them, which have coordination functions. In the second phase, bottom level automata receive orders from the higher levels and perform actions on the plant.

This mechanism is the same at each level of the hierarchy: given an automaton, the inferior levels send signals/communications to the automaton, which interprets such signals and communicates them to the superior levels; then, it receives orders from the superior levels that it processes and propagates to the inferior levels.

The consequence of this solution is that each computation phase performed by automata is necessarily divided in two distinct phases, called the upward phase and the downward phase. In the former, signals generated at the bottom level are progressively computed and propagated to the upper levels of the hierarchy, until the root of the hierarchy is reached. At this point, the downward phase is started and orders are processed and transmitted to the lower levels of the hierarchy; each automaton can send orders to several automata, while can receive orders only from one single automaton.

In practice, each automaton is associated with four sets of interface channels: \( \langle O, R, I, U \rangle \). With \( U \) we denote the set of upward output channels, i.e. the channels used to send signals/information to the upper level automata during the upward computation phase. With \( R \) we denote orders (also called requests) that can be received from an upper level automaton during the downward computation phase. With \( I \) we denote the set of input signals that can be received from lower level automata during the upward computation phase. Finally, \( O \) denotes the set of orders that can be sent to lower level automata during the downward computation phase.

The channels of set \( U \) of an automaton corresponds to set \( I \) of some higher level automaton; the same for channels \( R \), which are seen as in set \( O \) by a higher level automaton.

Example 1: An example of automaton hierarchy is depicted in Figure 1. With \( i_1, i_2, \ldots, i_7 \) we indicate input signals received from the plant, while with \( o_1, o_2, \ldots, o_7 \) we indicate actuator signals sent to the plant. All the upward arcs (i.e. \( c_1, c_4, c_5, c_7, c_9, c_11, c_12, c_13, c_14, c_16, c_18, c_19, c_20, c_22, c_24, c_26 \)) denote communications sent to the superior level automata during the upward computation phase. Conversely, all the downward arcs (i.e. \( c_2, c_3, c_6, c_8, c_10, c_15, c_17, c_21, c_23, c_25 \)) represent orders sent to inferior level automata; at most one downward arc enters each automaton.

The system works as follows. Input signals are captured from the plant by bottom automata (i.e. \( a_{51}, a_{52}, a_{53}, a_{61}, a_{62}, a_{63}, a_{64} \)). Then the latter send communication to the superior levels, which process the received information and send upward communication again, performing the upward computation. The upward computation phase ends when the root automaton \( a_0 \) is reached. At this point, the downward computation phase starts, and orders are sent by \( a_0 \) to \( a_1 \) and \( a_2 \). The two automata process the received orders and in turn send orders to the inferior levels. The order flow propagates...
to the bottom automata, which send orders to the plant. Terminated the downward computation, the process starts again with plant input scanning, upward computation, and so on.

Notice that automaton $a0$ has the responsibility of coordinating the entire system, and it sees two sub-systems, represented by $a1$ and $a2$. These two automata in turn coordinate other subsystems, and so on and so forth. Hence, the hierarchy corresponds to a distribution of responsibility: higher layers enforce strategic decisions and lower layers perform local processing.

2.2 Descriptive Layers

The automaton hierarchy certainly helps in defining a responsibility hierarchy in the system, but the design of the control system can be still hard. In order to simplify the design task, ASFA provides an important concept toward the modularization of the control system, i.e. the concept of descriptive layer.

A descriptive layer is a set of automata. Inside the layer, an automaton hierarchy still holds; thus, a layer can be seen as an autonomous sub-system, based on the same evaluation mechanism on which the overall ASFA network is based. If we see a descriptive layer from outside, it can be intended as a macro-automaton, with the same four types of channels/communications $(O, R, I, U)$, as simple automata. This way, there can be automata at higher levels which receive information from the descriptive layer and send orders to the layer, while the layer can receive signals from and send orders to automata at lower level in the overall hierarchy.

Furthermore, a descriptive layer can be contained in other descriptive layers; in practice, we can obtain a complex subsystem by composing several descriptive layers, each one devoted to perform a specific task, and so on.

Example 2: Consider again the ASFA network reported in Figure 1. The dotted areas show how descriptive layers can be introduced to modularize the design of the network. Consequently, if we want to hide the content of an inner descriptive layer, we can collapse it obtaining Figure 2.

2.3 Comparisons

Finite state models (FSM) have been in use for many years for specifying and implementing real-time applications which have to respond to events or stimuli. These events cause the system to move to a different state. "The problem with the state machine approach is that the number of possible states increases rapidly" (see [7]). To offset state explosion modularization mechanisms have been proposed for FSM, the best known possibly being the StateChart approach of [4]. Recently this approach has been incorporated into the Universal Modeling Language family of models [2].

Many diverse aspects of ASFA and of StateCharts could be compared but brevity limits the discussion to two relevant ones, modularization and concurrency. Modularization in Statecharts (and in UML state diagrams) is based on two mechanisms: the superstate and concurrent composition. A superstate collects a pool of states having some common transitions and hides the intra-pool state transitions.

Concurrent composition allows the global state to be obtained by two or more states taken from separate concurrent state machines. This mechanism is suitable to model systems having two or more sets of almost independent behaviors. Synchronization mechanisms between concurrent machines are rather delicate, because of the danger of causing deadlocks or losing signals.

The macroscopic difference between StateCharts and ASFA is that the former is a general purpose notation, the latter is specialized for plant automation systems. Thus ASFA imposes a strict discipline on the network topology, the signals exchanged by the automata, and periodicity of the input collection, decision making and output dispatching phases. Nothing of this kind exists in StateCharts.
3 Software Partitioning Problem

Once an ASFA network is defined, it is necessary to execute it in a real plant. A plant is generally supervised by several industrial PC, with multiprocessing capabilities.

Hence, it is necessary to generate from the ASFA network a set of processes, each responsible for executing a subset of the automata. This set of processes is obtained by the partitioning algorithm (see Section 4) to be presented that splits the ASFA network into blocks (called sequences), where each block of the partition corresponds to a process.

The requirements on the partition to be computed are the following. First, the partition must obey some static constraints (e.g. cohesion of a group of automata), defined by the ASFA designer. Second, the partition must be deadlock-free. Third, it should balance, as far as possible, the computational load among processes.

3.1 Graph Theoretical Model

We introduce the graph theoretical model for our problem.

**Definition 1: Automaton (Node)** An automaton is defined as a tuple \( a = (n, w_{up}, w_{down}) \), where \( n \) is the name of the automaton, \( w_{up} \) is the upward computational weight and \( w_{down} \) is the downward computational weight. □

A hierarchy \( H \) is defined on the ASFA network.

**Definition 2: Automaton Hierarchy** The Automaton Hierarchy is a set \( H = \{ h \} \). \( h \) is a tuple \( h = (n, l) \), where \( n \) is an automaton name and \( l \) is the hierarchy level at which the automaton appears.

The set \( H \) is correctly defined only if there is not a pair \( h_1, h_2 \in H \) such that \( h_1.n = h_2.n \) and \( h_1.l \neq h_2.l \). □

Based on the automaton hierarchy, each computation step of the ASFA network is then constituted by two distinct phases: the upward phase and the downward phase. The former proceeds in a bottom-up way: first of all, the automata at the bottom level of the hierarchy are processed, then those reached. The latter phase proceeds in the opposite sense.

**Definition 3: Communication (Arc)** A communication is described as a tuple \( c = (f, t) \), where \( f \) is the automaton which sends the communication, \( t \) is the automaton receiving the communication; automata \( f \) and \( t \) must belong to different levels of the automaton hierarchy \( H \).

An arc \( c \) is said an upward arc if \( t \) belongs to a higher level of the automaton hierarchy than \( f \), while is said a downward arc in the contrary case. In particular, only one single downward arc can enter a node. □

The above definition captures the structure of an ASFA network. Recall that an automaton is coordinated only by one automaton appearing at a superior level in the hierarchy, and may coordinate several automata appearing at an inferior level; but it cannot communicate with automata appearing at the same level of the hierarchy.

We revert to the graph representation of an ASFA network. In particular we consider two distinct graphs: the upward graph and the downward graph. This distinction is necessary, since the upward computation phase is distinct from the downward computation phase, the communications as well must be distinct.

**Definition 4:** The ASFA network is represented by a graph \( G = (V, E) \), where \( V \) is the set of nodes, each node corresponding to an ASFA automaton, while \( E \) is the disjoint union of \( E_{up} \) and \( E_{down} \), where \( E_{up} \) (respectively \( E_{down} \)) is the set of upward (resp. downward) edges (communications) connecting nodes in \( V \). We denote with \( G_{up} \) the upward graph \( G_{up} = (V, E_{up}) \), with \( G_{down} \) the downward graph \( G_{down} = (V, E_{down}) \).

Given a subset of nodes \( S \subseteq V \), this determines the sub-graphs \( G_{up}^S = (S, E_{up}^S) \) and \( G_{down}^S = (S, E_{down}^S) \), where \( E_{up}^S \subseteq E \) (resp. \( E_{down}^S \subseteq E \)) is the sub-set of upward (resp. downward) arcs connecting only nodes in \( S \). □

**Property 1:** The two graphs \( G_{up} \) and \( G_{down} \) are by definition acyclic. □

On the other hand the graph \( G \) may contain cycles. \( G_{up} \) describes the communications possibly occurring during the ASFA upward computation phase, while \( G_{down} \) describes the communications possibly occurring during the ASFA downward computation phase.

3.1.1 Constraints

On the ASFA network the designer can define additional constraints, which enforce hardware or application conditions that must be satisfied when executing the ASFA network.

Constraints are of two kinds: application constraints when it is the ASFA application itself that set them; hardware constraints when they are due to the architecture of the plant.

Consequently, the partitions outcome from the partitioning algorithm must respect such constraints.

**The Scheduling Flag Constraint.** This is a form of application constraint. A scheduling flag connects two or more
A practical situation in which the scheduling flag is useful is when a sub-system can operate in two distinct modes, for instance in diagnostic mode versus operational mode: in the former case, the task of the sub-system is to check if the plant works correctly, in the latter the task of the sub-system is to actually let the plant execute a command. Depending on the rest of the system, the sub-system may alternatively be forced to operate either in diagnostic or in operational mode.

**Definition 5: Scheduling Flag Constraint**  
Let $S_1, S_2, \ldots, S_n$ several sub-graphs such that for each pair $S_i, S_j$ with $i \neq j$, it is $S_i \cap S_j = \emptyset$. A scheduling flag $f$ is a function $f : \{1, 2, \ldots, n\} \to \{S_1, S_2, \ldots, S_n\}$, such that $f(i) = S_i$, with $i \in \{1, 2, \ldots, n\}$. With $F = \{f_i\}$ we denote the set of scheduling flag constraints. $\Box$

**Interference Constraint.** The first type of hardware constraint that it is possible to specify for an ASFA network is the interference constraint. Such a constraint involves two sub-graphs $S_1$ and $S_2$, and imposes that the automata in $S_1$ and those in $S_2$ cannot be processed on the same processor.

The interference constraint comes out from the architecture of the control system: different processors generally controls specific parts of the plant, parts that may be far away from each other. Therefore, the two ASFA sub-graphs devoted to control such two parts, necessarily cannot be executed on the same processor.

Furthermore, in order to improve the reliability of the plant, a clever use of the interference constraint may avoid that a single failure affects two or more system critical functions, that are charged to the same physical node.

**Definition 6: Interference Constraint** Consider two sub-graphs $S_1$ and $S_2$, such that $S_1 \cap S_2 = \emptyset$. An interference constraint is described by a pair $Ic = \langle S_1, S_2 \rangle$. The set of interference constraints is denoted as $I = \{Ic\}$. $\Box$

**Cohesion Constraint.** A cohesion constraint is another hardware constraint, which specifies a cohesion property on a sub-graph: when such a constraint is defined, the automata belonging to the specified sub-graph must be assigned to the same process. Furthermore, the ASFA designer can also specify an atomicity property for a sub-graph $S$ on which a cohesion constraint holds: if set, such a property denotes that no automata except those belonging to $S$ can be computed by the process computing the automata in $S$.

**Definition 7: Cohesion Constraint** A cohesion constraint is a pair $Cc = \langle S, a \rangle$, where $S$ is the sub-graph on which the constraint holds, while $a$ is a flag ($a \in \{true, false\}$) indicating if the atomicity property is required. We indicate with $C = \{Cc\}$ the set of cohesion constraints $Cc$. The set $C$ is correctly defined only if, for each pair $Cc_i, Cc_j \in C$ with $i \neq j$, it holds $Cc_i.S \cap Cc_j.S = \emptyset$. $\Box$

From a certain point of view, a cohesion constraint is motivated by opposite needs w.r.t. the interference constraint. In particular, we can have a sub-graph $S$ for which the scheduling flag holds, which might contain a sub-graph $S' \subseteq S$ for which the cohesion constraint is specified.

The same considerations hold for two sub-graphs $S_1$ and $S_2$ under an interference constraint: the automata in $S_1$ (resp. $S_2$) must not be processed by the same processes computing automata in $S_2$ (resp. $S_1$), but the computation of automata in $S_1$ (resp. $S_2$) can be performed by several parallel processes. Furthermore, on a sub-graph $S' \subseteq S_1$ (resp. $S'' \subseteq S_2$) it is possible to specify a cohesion constraint.

### 3.2 A Priori Partitioning

Having defined the graph theoretical model, we move to the problem of partitioning the ASFA network into parallel processes. To do that, it is necessary to clarify how automata are executed by a process.

**Definition 8:** A sequence $S = (s_1, s_2, \ldots, s_n), s_i \in V$ is an ordered set of automata. $\Box$

A process $p$ is associated to a sequence $S$: in particular, $p$ operates as follows.

1. **Upward inputs.** The process waits until all the inputs concerning the upward computation of the automata in $S$ and coming from automata in other processes are ready.
2. **Upward computation.** Received all the external upward inputs, the process executes the upward computations of the automata in $S$, in the order in which the automata appear in $S$. During the computation, the automata may generate outgoing communications.
3. **Downward Inputs.** The process waits until all the inputs concerning the downward computation of the automata in $S$ and coming from automata in other processes are ready.
4. **Downward computation.** Received all the external downward inputs, the process executes the downward computations of the automata in $S$, in reverse order. The computed automata may generate outgoing communications.

Terminated the downward computation, the process starts again form step 1. Note that steps 1 and 3 waits only for external communications: communications that involve only automata in the sequence are managed internally at the process. The only constraint is that the order in the sequence respect the topological sort induced by the upward communications, while the reverse sequence respect the topological sort induced by the downward communications.
3.3 Problem Definition

The problem is defined on the concept of valid partition.

**Definition 9: Valid Partition** Given an ASFA network $G = (V, E)$, the set $F$ of scheduling flag constraints, the set $I$ of interference constraints, and the set $C$ of cohesion constraints, a valid partition $P$ is the set $P = \{S_i\}$ of sequences for which the following conditions hold:

- (Deadlock) - Taken two sequences $S_i, S_j \in P$, and four automata $a_1, a_2, a_3, a_4$, with outward communications $\langle a_1, a_2 \rangle$ and $\langle a_2, a_3 \rangle$. Suppose now that the three automata are allocated into the two sequences $S_1 = (a_1, a_3)$ and $S_2 = (a_2)$.

- If this is the situation, the two processes $p_1$ and $p_2$, processing the sequences $S_1$ and $S_2$, respectively, are in deadlock. In fact, $p_1$ cannot start, since the automaton $a_3$ is waiting for inputs coming from the automaton $a_2$, computed by $p_2$. However, $p_2$ cannot start, since the automaton $a_2$ is waiting for inputs coming from $a_1$ computed by $p_1$.

4 The Partitioning Algorithm

A valid partition $P$ must meet Definition 9, but among all the valid partitions, a good one should show a good degree of parallelism and, if possible, a balanced computational load.

Hence, we propose an algorithm that, by means of an heuristic method, given an ASFA network $G$ generates a partition $P$ which approximates the desired characteristics.

4.1 Basic Sub-nets

The scheduling flag constraints defined on an ASFA network induce a preliminary sub-division of the network into net appearing in the co-domain of some scheduling flag function, or the network obtained removing all the sub-nets which are in the co-domain of some scheduling flag function.

This is motivated by the mutually exclusive execution of sub-nets in the co-domain of a scheduling flag function: automata in two mutually exclusive sub-nets must be computed by two distinct sets of sequences, each one focused only on automata appearing in one of the two sub-nets. This way, sequences which process a not activated sub-net are not at all activated, and only sequences processing automata in the activated network will run. Furthermore, cohesion constraints with the atomicity parameter set to True induce basic sub-nets; However, since no other automata can be processed by the process which computes the sub-net specified by the constraint, these are not considered for further partitioning.

**Definition 10: Basic Sub-net** Given an ASFA network $G = (V, E)$, a basic sub-net $b$ is a tuple $b = (V_b, E_b)$, where $V_b \subseteq V$ and $E_b \subseteq E$, such that an arc $\langle a_i, a_j \rangle \in E_b$ iff both $a_i, a_j \in V_b$. □

4.2 The Algorithm

Basic sub-nets are directly obtained by the scheduling flag constraints. Furthermore, the user normally wishes to further partition such sub-nets; in general, for each sub-net the user specifies the parameter min, i.e. the minimum number of sequences into which to partition the basic sub-net.

We now introduce the partitioning algorithm. The algorithm makes use of the concept of Collapsed Sub-net.

**Definition 11: Collapsed Sub-net** A Collapsed Sub-net $\overline{G}$ is a tuple $\overline{G} = (\overline{V}, \overline{E})$ such that $\overline{V}$ is the set of collapsed nodes, and $\overline{E}$ is the set of arcs. A collapsed node $\overline{v}$ is a tuple $\overline{v} = (N, w_{up}, w_{down})$, where $N$ is the set of automata collapsed into $\overline{v}$, while $w_{up}$ and $w_{down}$ are respectively the upward and downward computational weights of the collapsed nodes. □

In practice, a collapsed sub-net is the result of an aggregation process, which aggregates several automata into one single node. Each collapsed node has associated the set of collapsed automata $N$.

4.2.1 The main algorithm

The main algorithm is reported at the end if this section. Each iteration of the algorithm performs aggregations over the collapsed sub-net obtained by the previous iteration or, in the case of the first iteration, over the collapsed sub-net directly derived by the basic sub-net to be partitioned (line 1).

At each iteration (identified by $k$ and described by lines 4 to 22), the algorithm tries to aggregate nodes by topologically sorting them. Each iteration considers in particular one of the ASFA computation phases: variable phase alternatively is set to Up or Down (lines 3 and 22). The aggregation task is performed by the inner loop (lines 9 to 16). The topological sort is computed by means of the function named
The algorithm is based on a topological sort strategy to prevent deadlock. In fact, aggregations that generate cycles in the collapsed sub-net are not considered at all; this is ensured by the way function AdmissiblePaths selects paths from the working sets of nodes and arcs.

However, not every path which is in principle admissible is selected: function AdmissiblePaths receives a parameter, named th\textsubscript{max}, which is the maximum threshold for weights of admissible paths: a path is admissible only if its weight in the current phase is less than or equal to the threshold. The threshold th\textsubscript{max} has been introduced for balancing the computational loads among parallel sequences. It is defined as

\[ th_{\text{max}} := \frac{\left(\sum_{p \in \mathcal{V}_k} \pi_p \cdot w_{\text{phase}}\right)}{|\mathcal{V}_k|/2} \]

which is motivated by the following reasons. At the end of each iteration, we want to obtain a collapsed sub-net which contains no less than the half of nodes of the sub-net on which the iteration have to work. Then with the term \((|\mathcal{V}_k|/2)\) we denote the minimum number of nodes to obtain. On the other side, the term \((\sum_{p \in \mathcal{V}_k} \pi_p \cdot w_{\text{phase}})\) denotes the overall weight of the sub-net in the current phase. Hence, the threshold is the maximum weight a collapsed node should have in the new collapsed sub-net.

The inner loop works as follows. If the set A of paths extracted by function AdmissiblePaths is not empty, or the minimum number of collapsed nodes has not been reached yet, a path p from A is selected (line 11) and the corresponding collapsed node \(\pi\) is added to the new set of nodes \(\mathcal{V}_{k+1}\) (line 11). Consequently, all nodes in p must be removed from the working set of nodes (line 13) and all arcs concerning nodes in \(\pi\) must be removed from the working set of arcs (line 14); this way, the next call of function AdmissiblePaths will operate on a reduced graph. Furthermore, p is removed from A (line 15) and, if A is now empty, function AdmissiblePaths is called again.

When the inner loop terminates, if the minimum number of collapsed nodes has not been reached yet, at line 18 function {\textit{CollapsedIsolated}} removes from \(\mathcal{V}_{k+1}\) i, if present, isolated nodes, aggregating them to other nodes in the sub-nets. The function considers only those nodes whose weight is less than the threshold \(th_{\text{max}}/A\); this is due to the necessity of keeping balanced the computation load, and aggregate isolated nodes only when their weight does not make unbalanced the sub-nets (observe that the value of \(th_{\text{max}}\) increases}
11 choose a path $p$ from $A$ and create a collapsed node $\pi$ from $p$
12 $\bar{V}_{k+1} := \bar{V}_{k+1} \cup \{\pi\}$
13 delete from $\bar{V}'$ all nodes in $p$
14 delete from $\bar{E}'$ all arcs connecting nodes in $p$
15 delete $p$ from $A$
16 if $|A| = 0$ then $A := AdmissiblePaths(\bar{V}', \bar{E}', t_{h_{\max}}, phasew, I)$
end
17 if $|A| \neq 0$ then $\bar{V}_{k+1} := \bar{V}_{k+1} \cup \bar{V}'; conv := true$
18 if $conv = false$ then $conv :=$
19 CollapseIsolated($\bar{V}_{k+1}, min, t_{h_{\max}}/4$)
20 $\bar{E}_{k+1} :=$ the set of arcs in $bs.\bar{E}$
21 connecting nodes in $\bar{V}_{k+1}$
20 if $|\bar{V}_{k+1}| = |\bar{V}_k|$ then $conv := true$
21 $k := k + 1$
22 if $phasew = Up$ then $phasew := Down$
23 else $phasew := Up$
end
23 until $conv = true$
24 return $\tilde{G} = (\bar{V}_k, \bar{E}_k)$
end

4.2.2 Computing admissible paths

Function $AdmissiblePaths$ (reported at the end of this section) actually searches for paths in the working set of nodes. When the function is called, it receives only the set of nodes (and the corresponding set of arcs) that have not been collapsed. It also receives the set of interference constraints, since it must not aggregate automata that, based on an interference constraint, must be necessarily processed by two distinct processes.

At line 1, the result set $A$ of admissible paths is set to empty; then, at line 2, variable $opp$ is set to denote the opposite phase w.r.t. the current phase indicated by parameter $phasew$. Then line 3 computes the set $P$ of paths over the working graph, such that each path has not incoming arcs of the current phase (line 5), has not outgoing arcs of the opposite phase (line 6), and does not contain nodes for which an interference holds (line 7). In the actual implementation, this work is performed by means of a topological-sort based procedure. Observe that the constraints specified at lines 5 and 6 are motivated by the need of avoiding deadlocks: in fact, if the constraints are met, the risk of deadlock is avoid, because we exclude the possibility of circularity.

At this point, the function extracts from $P$ the set $O$ of singletons with weight greater than the threshold $t_{h_{\max}}$, because in the following lines only paths with weight less then the threshold will be extracted. However, the threshold $t_{h_{\max}}$ is valid only for paths, and singletons must be anyway selected for continuing the partitioning process.

Among all the singletons, line 10 chooses only one of them and uses it to generate the result set, which contains only the singleton.

In contrast in the case there are not singletons with high weight, the else branch of line 12 operates. Line 13 discards from $P$ all paths with weight greater than the threshold $t_{h_{\max}}$. If the resulting set $P$ is not empty (line 14), the path $p_{\max}$ with maximum weight (which is also less than the threshold) is selected (line 15) and inserted into the result set $A$ as the first path (line 16). However, among all the

Figure 3. Sample iteration of the algorithm.

Figure 4. Sample iteration of the algorithm (2).
To make clear how the algorithm works, consider the working graph depicted in Figure 3.a. In the figure, at the right side of a node label we report the computational weight of each node. Upward communications are denoted by solid arcs, while downward communications are denoted by dashed arcs. The overall computational load is 8, then the value of the threshold $t_{\text{max}}$ is 2.6.

We now simulate an iteration, working with variable phase set to Up. The first call of function $\text{AdmissiblePaths}$ selects the singleton $a_0$, since its weight is greater than $t_{\text{max}}$; hence, the new collapsed node $b_1$ is inserted into the node set $\overline{V}_{k+1}$ (Figure 3.a). The second call of function $\text{AdmissiblePaths}$ selects two paths (see Figure 3.b): the first one, labeled as $\alpha$ in the figure, is the path with maximum weight and less than $t_{\text{max}}$; the second path (labeled as $\beta$ in the figure), is a path independent of $\alpha$ and with weight greater than $t_{\text{max}}/3$, then it can be chosen (observe that this way only independent paths with a somehow comparable weight are selected to work in parallel). Consequently, two collapsed nodes $b_2$ and $b_3$ are inserted into $\overline{V}_{k+1}$. At this point (Figure 4.c) only two nodes remain in the working graph; they are both selected by function $\text{AdmissiblePaths}$ because $a_1$ is the heaviest nodes, while $a_2$ constitutes an independent path with sufficient weight.

Finally, Figure 4.d shows the graph for the next iteration. Observe that in the upward phase node $b_3$ can be executed in parallel with the rest of the network; node $b_4$ can be executed in parallel with node $b_5$, but both must wait that $b_2$ terminates. In the downward phase, nodes $b_4$ and $b_5$ are independent, while nodes $b_5$, $b_3$ and $b_2$ are serialized.

## 5 Evaluations

In order to evaluate the algorithm, we performed experiments on a case study derived from a real application, an ASFA network that supervises a power station. The net-
The last time the function was called, it produced 11 sequences, one less than the requested number. However, the deadlock prevention and the degree of parallelism (maximum number of sequences that can work in parallel) are significantly limited.

The algorithm obtains the desired number of sequences. It produces the result performing 18 iterations. Due to the partitioning, the number of communications (both upward and downward) are collapsed into the same sequence. From Figure 6 we obtain an idea of the distribution of computational loads among sequences. The values of variance and average load which characterize the partitioned network are coherent with such a situation, meaning that the algorithm is effective in keeping the computational load balanced among the two computational phases. Finally, as far as the maximum parallelism is concerned, 4 sequences can work in parallel both in the upward and downward computation phases. However, the deadlock prevention and intricate communications which characterize the case study strongly limit the maximum degree of parallelism.

Consider now the second trial. We asked the algorithm to partition the network into 12 sequences. The algorithm produced 11 sequences, one less than the requested number. This is due to the greedy strategy adopted by function AdmissiblePaths; the last time the function was called, it selected a path in such a way remaining nodes in the working set were not sufficient to reach 12 sequences. Apart from a further significant reduction of communications among sequences (that could be easily expected) it is interesting to notice that only one more iteration is necessary w.r.t. the previous trial. This confirms that the strategy on which the algorithm is based is rather effective. As far as the computational loads are concerned, the results are still interesting: both the variance and the average load reflect the fact that the downward computational load is twice the upward computational load. In contrast, the maximum parallelism reduces to 3 for the upward computational phase and to 2 for the downward computational phase.

We run three experiments: at first, we tried to partition the network into 18 sequences, then we tried with 12 sequences and finally we tried with 6 sequences. Figure 5 reports some data about the structure of the obtained sequences and the number of iterations performed by the algorithm; Figure 6 reports data concerning the distribution of the computational loads and the degree of parallelism (maximum number of sequences that can work in parallel).

We start examining the results of the first trial, i.e. partitioning into 18 sequences. At first, we notice that the algorithm obtains the desired number of sequences. It produces the result performing 18 iterations. Due to the partitioning, the number of communications (both upward and downward) significantly reduces (around a factor 10); this is due to the fact that many communicating automata are collapsed into the same sequence. From Figure 6 we obtain an idea of the distribution of computational loads among sequences. In the considered network, the downward phase is heavier (twice) than the upward phase. The values of variance and average load which characterize the partitioned network are coherent with such a situation, meaning that the algorithm is effective in keeping the computational load balanced among the two computational phases. Finally, as far as the maximum parallelism is concerned, 4 sequences can work in parallel both in the upward and downward computation phases. Observe that this number may seem too small, if compared with the number of sequences. However, the deadlock prevention and intricate communications which characterize the case study strongly limit the maximum degree of parallelism.

Consider now the second trial. We asked the algorithm to partition the network into 12 sequences. The algorithm produced 11 sequences, one less than the requested number. This is due to the greedy strategy adopted by function AdmissiblePaths; the last time the function was called, it selected a path in such a way remaining nodes in the working set were not sufficient to reach 12 sequences. Apart from a further significant reduction of communications among sequences (that could be easily expected) it is interesting to notice that only one more iteration is necessary w.r.t. the previous trial. This confirms that the strategy on which the algorithm is based is rather effective. As far as the computational loads are concerned, the results are still interesting: both the variance and the average load reflect the fact that the downward computational load is twice the upward computational load. In contrast, the maximum parallelism reduces to 3 for the upward computational phase and to 2 for the downward computational phase.

Finally, we tried to get down to 6 sequences. We notice that the algorithm is still able to reach the desired number of sequences and it needs only one more iteration than the previous trial. Even the computational loads seem to be well balanced: if we look at the variance, we observe that the load is better distributed that in the previous trial. However, the algorithm fails in obtaining parallelism.

6 Conclusions

The proposed network partitioning algorithm is a practical tool for decomposing a large network of automata into a small number of sub-nets, to be allocated to the available computers. A previous prototype was quite good in balancing the computational load of the sub-nets but did not consider the localization constraints imposed by the designer and incurred in occasional deadlocks. The new algorithm will contribute to the reengineering of power plant automation systems in view of their porting on modern distributed systems. Our scheduling algorithm is an original development of classical ideas for the unusual but relevant case of hierarchical two-sweep systems. It is hoped that the conceptual approach, if not the specific criteria, could provide inspiration to other hierarchical systems.

References


