9-24-2004

Decomposing controllers into non-conflicting distributed controllers

Padmanabhan Krishnan
Bond University, Padmanabhan_Krishnan@bond.edu.au

Decomposing Controllers Into Non-Conflicting Distributed Controllers [1]

Padmanabhan Krishnan
Centre for Software Assurance
Faculty of Information Technology
Bond University
Gold Coast, Queensland 4229, Australia
Email:pkrishna@staff.bond.edu.au

Abstract. In this article we present an application of decompositions of automata to obtain distributed controllers. The decomposition technique is derived from the classical method of partitions. This is then applied to the domain of discrete event systems. We show that it is possible to decompose a monolithic controller into smaller controllers which are non-conflicting. This is derived from the notion of decompositions via partitions. Some global state information is necessary to ensure that the joint behaviour of the component automata is identical to the original controller. The global state information required is identical to the global information present in Zielonka asynchronous automata. The joint behaviour of the component automata is shown to be non-conflicting.

Keywords: Decompositions, Asynchronous Automata, Controllers

1 Introduction

It is well known that one can synthesise controllers for discrete event systems (DES) using von-Neumann discrete game playing techniques [RW89]. In the context of automata one is given a system description (also called a plant or environment) and a specification of desired behaviour. The synthesis process generates an automaton called the controller or supervisor. The synthesis process ensures that the joint behaviour of the controller and the plant is within the behaviours stated in the specification. The principal advantage of this synthesis process is that it is completely automatic. Furthermore, the synthesised controller is the most general controller (i.e., the controller that permits the largest possible behaviour). In this article we present an approach to synthesising distributed/parallel controllers.

The modular approach to developing (including specification, refinement, verification) systems is the most promising technique to overcoming complexity [Jon94b,dRLP98]. Compositional techniques allow one to combine smaller systems to obtain larger ones. But it imposes certain conditions to ensure that the large system satisfies the requirements. While there is knowledge concerning general compositional ideas [dRLP98], the situation for controller synthesis is not that clear. [WR88] identify a sufficient condition called *non-conflicting* under which the joint operation of two controllers is valid. The literature [Won04] indicates that modular synthesis of controllers is difficult and not

[1] Part of this work was supported by Siemens Research, Munich, Germany.

always successful. [SW04] show how the state space of a single controller can be reduced but do not address the distributed case. The general synthesis problem especially related to implementation is either undecidable or NP-complete [SEM03]. [PTV01] also show how decentralised control over partial observations does not always admit finite state controllers (although the overall system may be finite state). This is because the projection on the individual observations may yield non-regular languages. Tripakis in [Tri04] shows that determining the existence of a function that determines the validity of a distributed observation is undecidable. This result is related to the undecidability of a decentralized supervisory control problem. Therefore, the next best thing is to perform a "monolithic" synthesis and then automatically decompose the controller to obtain distributed controllers. By obtaining a suitable distributed automata over different alphabets, this process can also be viewed as yielding a decentralised controller over partial observations. It is this problem that we solve in this paper.

Our solution to this problem is presented in two stages. In the first stage we treat the controller obtained from the monolithic synthesis as an independent entity (i.e., an open system). We adapt the classical decomposition technique [HS66] to split the large controller into two components along with synchronisation restrictions. The key result is that this process yields non-conflicting controllers. In the second stage we simplify the two controllers (i.e., relax the synchronisation requirements) by taking the behaviour of the plant into account. While this affects the overall behaviour of the controller, it does not alter the correctness of the controller working in conjunction with the plant. This technique illustrates how a collection of simple controllers each controlling only certain aspects of behaviour but achieving the overall control can be constructed. The strategy described in this paper can be summarised as follows.

1. Generate modular controllers from the distributed system.
2. If the generated controllers are non-conflicting then no further action is necessary.
3. Otherwise, generate a monolithic controller.
4. Decompose the monolithic controller to obtain two non-conflicting controllers. This is described in section 3 and an example is presented in section 4.
5. Simplify the two controllers using the behaviour of the plant. This is described in section 5.

In the next section we review the relevant results pertaining to decomposition of automata using partitions, relevant definitions from DES and the definition of distributed automata. We then introduce a different presentation of distributed automata. This is followed by the main results of this paper. First we show the decomposition of a single automaton into distributed automata such that the languages accepted by them are the same. Then we show that the decomposition process using partitions ensures non-conflicting behaviour. Two examples, one of which is devoted to the decomposition of controllers, are then presented. Finally, we present the extension to the basic technique which takes into account the behaviour of the plant that is being controlled.

2 Preliminaries

In this section we briefly recall the concepts and notation required to explain decompositions and controllers. The details concerning decompositions can be found in [HS66,Hol82] while the details concerning discrete event systems and controllers can be found in [RW89].

2.1 Decompositions

A finite state automaton $\mathcal{A}$ consists of a 5 tuple $(Q, \Sigma, \longrightarrow, q^0, F)$ where Q is a finite set of states, Σ a finite input alphabet, $\longrightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, $q^0 \in Q$ the initial state and $F \subseteq Q$ the set of final states. As a notational convenience $(q, a, q') \in \longrightarrow$ is often written as $q \xrightarrow{a} q'$.

We extend the transition relation to sets. That is, for $(X \subseteq Q)$ and $(a \in \Sigma)$ we let $X \xrightarrow{a} Y$ where $Y = \{q' \mid (q \in X), (q \xrightarrow{a} q')\}$. Given two automata $\mathcal{A}_1=(Q_1,\Sigma,\longrightarrow_1,q_1^0,F_1)$ and $\mathcal{A}_2=(Q_2,\Sigma,\longrightarrow_2,q_2^0,F_2)$ we write $(\mathcal{A}_1 \leq \mathcal{A}_2)$ if there is a *surjective partial* map η from Q_2 to Q_1 satisfying the following three properties: 1) $\forall\ q,q' \in Q_2$, $a \in \Sigma$: $\eta(q) \xrightarrow{a}_1 \eta(q')$ implies $q \xrightarrow{a}_2 q'$, 2) $\eta(q_2^0) = q_1^0$ and 3) $F_2 = \{q \in Q_2 \mid \eta(q) \in F_1\}$

That is, there is a *covering homomorphism* from $\mathcal{A}_2$ to $\mathcal{A}_1$. It is possible for $\mathcal{A}_2$ to have more states and transitions than $\mathcal{A}_1$. However, if $\mathcal{A}_2$ has no a move in a state q, there cannot be an a move from $\eta(q)$ or $\eta(q)$ is undefined.

Given two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ over a common input alphabet Σ, define $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ to represent the *synchronous product* (also called the meet operator in [Won04]). That is, the automaton $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ is defined as $(Q_1 \times Q_2, \Sigma, \longrightarrow, q_1^0 \times q_2^0, F_1 \times F_2)$ where $(q_1,q_2) \xrightarrow{a} (q_1',q_2')$ if and only if $q_1 \xrightarrow{a}_1 q_1'$ and $q_2 \xrightarrow{a}_2 q_2'$. We say $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ is a *decomposition* of $\mathcal{A}$ if and only if $\mathcal{A} \leq (\mathcal{A}_1 \parallel \mathcal{A}_2)$. A decomposition of $\mathcal{A}$ into $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ is *non-trivial* if $\mathcal{A}$ is not identical to $\mathcal{A}_1$ or $\mathcal{A}_2$.

Given $\mathcal{A}$, a partition π over Q is *admissible* if and only if for every X belonging to π and for every a in the input alphabet, there is a Y belonging to π such that $X \xrightarrow{a} Z$ and $Z \subseteq Y$.

The product of two partitions π_1 and π_2 written as $\pi_1 \cdot \pi_2$ is defined as follows. $\pi_1 \cdot \pi_2 = \{\, X \cap Y \mid X \in \pi_1, Y \in \pi_2, X \cap Y \neq \emptyset\}$. The finest partition $\perp_Q$ is defined to be set of all singletons of Q. The coarsest partition $\top_Q$ is the set $\{$ Q $\}$. A partition is non-trivial if it is neither the coarsest nor the finest partition. Two partitions are *orthogonal* if their product yields the finest partition.

Proposition 1 ([Hol82]). If a given automaton $\mathcal{A}$ has two non-trivial orthogonal partitions each of which is admissible, then it has a non-trivial decomposition.

The proof of the proposition constructs $\mathcal{A}_1$ and $\mathcal{A}_2$ by considering orthogonal admissible partitions. That is, each state in the component automata is actually a subset of states of the original automaton. The construction also ensures that $\mathcal{A} \leq (\mathcal{A}_1 \parallel \mathcal{A}_2)$. In such a case, $\mathcal{A}$ is said to be *decomposed via partitions.*

2.2 Non-Conflicting Controllers

Given a description of a plant (say P) (or the environment) and a desired specification (say S), the purpose of a controller (say C) is to ensure that the operation of the plant is within the desired specification. That is, the behaviour of (P || C) should be contained in S. There are certain restrictions on C for it to be called a controller for P. Every symbol in the alphabet of the automaton P can be classified as either controllable or uncontrollable. A controllable action can either be disabled or enabled by the controller. An uncontrollable action cannot be disabled to the controller. Hence in any global state of (P || C), if P can exhibit an uncontrollable action, C must also be able to exhibit it.

This can be formally stated as follows. Given an alphabet Σ, let Σ_u denote the uncontrollable subset of Σ and Σ_c denote the controllable subset of Σ. A prefix closed language L is a *controllable language* with respect to a prefix closed language G if and only if for every $\alpha \in \Sigma^*$ belonging to L and a belonging to Σ_u, if αa belongs to G, αa belongs to L. Here G represents the trace behaviour of the plant P and L the trace behaviour of the controller C. In any given state of the computation (denoted by α), if the plant can perform an uncontrollable action (a) the controller must allow it to occur. The above definition can be extended to non-prefix closed languages. A language L is a controllable language with respect to a language G iff the prefix closure of L controllable with respect to the prefix closure of G.

The controller itself may consist of a number of parallel components, and the key idea is that they should not be conflicting one another. Given a regular language L, let $\tilde{L}$ represent the prefix closure of L. Two languages L and K are *non-conflicting* iff $\tilde{L} \cap \tilde{K}$ is identical to $\widetilde{L \cap K}$. It is easy to see that $\widetilde{L \cap K} \subseteq \tilde{L} \cap \tilde{K}$. So, to verify non-conflcting behaviour, we only have to show inclusion in one direction. The usefulness of non-conflcting behaviour is illustrated by the following proposition.

Proposition 2 ([WR88]). *Let L_1 and L_2 be supremal (largest with respect to inclusion) controllable sub-languages of E_1 and E_2 respectively. If L_1 and L_2 are non-conflicting controllable languages , $L_1 \cap L_2$ is the supremal controllable sublanguage of $E_1 \cap E_2$.*

We now give an example to illustrate the problem of conflcting controllers or supervisors.

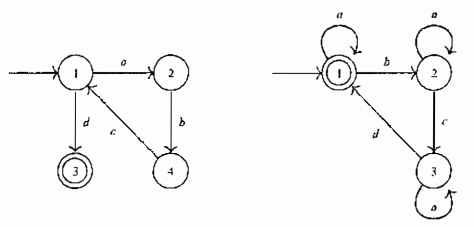

Fig. 1. Conflicting Automata

Consider the two automata shown in figure 1. Let L be the language accepted by the automaton shown on the left and K be the language accepted by the automaton shown on the right. Note that $L \cap K = \{abcd\}$. Consider the string *abca* which belongs to ($\tilde{L} \cap \tilde{K}$). This is because the string *abcabcd* belongs to L and the string *abcad* belongs to K. Hence the string *abca* belongs to the intersection of the prefix closures of the two languages. However, it cannot belong to the prefix closure of ($L \cap K$). Such behaviour is conflicting because if the environment asks for permission to exhibit the symbol a after exhibiting abc, both controllers give their permission. But the system becomes stuck as no extension of this string can lead to a final state. Hence the system enters a deadlock state. This arises as the automaton is viewed as interacting with an environment rather than as a pure acceptor. In other words, non-conflicting controllers ensure the plant does not deadlock under their joint supervision.

2.3 Asynchronous Automata

We now describe asynchronous automata [Zie87]. We assume a finite set $Loc = \{1,\dots,n\}$ of agents. Associated at each location is an automaton with its corresponding alphabet. These are combined as follows. Let $(\Sigma_1,\Sigma_2,\dots,\Sigma_n)$ be a distributed alphabet and $(\mathcal{A}_1,\mathcal{A}_2,\dots,\mathcal{A}_n)$ be the associated automata. We will assume that the state spaces of the component automata are pairwise disjoint but could have overlapping alphabets.

Let $\Sigma = \bigcup_{i\in Loc} \Sigma_i$. For $a \in \Sigma$ let $loc(a) = \{j \mid a \in \Sigma_j\}$. We define $Q = \bigcup_{i\in Loc} Q_i$ to be the set of local states. Given $L \subseteq Loc$, we define the set of possible L-states, $Q_L = \{ q\colon L \longrightarrow Q \mid \forall l \in L, q(l) \in Q_l \}$. Q_L defines the global state of the automaton as viewed from locations in L. For every action $a \in \Sigma$ we write Q_a to represent $Q_{loc(a)}$. Similarly, for $q \in Q_{Loc}$, we let q_a represent the restriction of q to $loc(a)$ and q_{-a} represent the restriction of q to $Loc - loc(a)$.

A set of a-transitions (indicated by $\Rightarrow_a \subseteq Q_a \times Q_a$) is defined for every q and q' belonging to Q to be $(q,q') \in \Rightarrow_a$ implies that for every l belonging to $loc(a), q(l) \xrightarrow{a}_l q'(l)$.

These are the a moves or the moves made by the automata to exhibit a. Strictly speaking the individual transitions are not necessary, but they help in the presentation of the automata. If one is given only $\Rightarrow_a$, the local transitions can be derived. The global transition relation on Q_{Loc}, written as $q \xrightarrow{a} q'$ is defined to be $(q_a,q'_a) \in \Rightarrow_a$ and $q_{-a} = q'_{-a}$.

Assume a global initial q^0 and a set F of final states. That is, for every l, $q^0(l)$ is identical to to q^0_l. Similarly, if f belongs to F, for every l, $f(l)$ belongs to F^l. The Zielonka asynchronous automaton is given by the 5-tuple $(Q_{Loc},\Sigma,\longrightarrow,q^0,F)$. At this stage it is relevant to note that the asynchronous automata are not the same as the taking the sychronous product of the individual automata. In asynchronous automata not all states in the product space where the an action is possible individually is necessarily enabled. As shown in [Zie87] the language $((aa \mid bb)c + (a \mid b)c)^*$ where $\mid$ indicates the shuffle cannot be accepted by a product of two automata over the alphabet $\{a,c\}$ and $\{b,c\}$. An asynchronous automaton can be defined to accept this by defining the global states where a c can be exhibited to be either after one a and one b or after two as and two bs.

We use a new presentation of asynchronous automata called *restricted product automata*. The new representation is useful in the synthesis process. The notion of local and global states is as before. Instead of having the family of transition relations $\Rightarrow_a$, we have a family of synchronisation constraints. That is, the states in which the action a is possible is explicitly defined. For each input symbol a, we assume a set $synch_a$ which is contained in Q_a such that if the cardinality of $loc(a)$ is 1, $synch_a$ is identical to Q_a.

The interpretation is that an action a can be exhibited only from a permitted a-state. Such a set is defined for each action in the alphabet. This family of sets represents the global synchronisation information. The requirement on $synch_a$ states that purely local actions do not depend on any global information and hence cannot be disabled. The global transition relation, $\longrightarrow$, can now be defined as follows.

We let $(q \xrightarrow{a} q')$ if and only if the three conditions $q_l \xrightarrow{a}_l q'_l$ for all $l \in loc(a)$, $q_{-a} = q'_{-a}$ and $q_a \in synch_a$ hold.

The first two conditions are similar to that of Zielonka automata. The last condition states that the action a can be exhibited only from a permitted state; or in other words an action is allowed in state q_a. In synchronising automata described in [Ram95], the component automata can have common states. If a common action is exhibited, the resulting states have to be identical. As an aside, the difference with synchronising automata in [Ram95] is that we require the states to "agree" before the action rather than after the action.

The following proposition makes it clear that restricted product automata are only a new presentation of asynchronous automata. That is, we are using syntactic sugar to simplify the synthesis process.

Lemma 3. *Deterministic Zielonka asynchronous automata correspond exactly to deterministic restricted product automata.*

Proof: By converting every a-transition to transitions for the component automata and by including the a-state into the set of permissible states we can translate one automaton into another. That is, $(q,q') \in \Rightarrow_a$ iff q belongs to $synch_a$. Furthermore, for every l belonging to $loc(a)$, we demand $q(l) \xrightarrow{a}_l q'(l)$ •

While the theory for asynchronous automata is presented in terms of some finite (n) components, we now restrict our attention to two component system. This is primarily because the decomposition theory yields two components. The approach presented here can be iterated to obtain any number of component systems.

3 Exact and Non-Conflicting Decompositions

Let $\mathcal{A}$ be a controller synthesised from some plant description along with the specified behaviour. The task is to decompose this automata into two non-conflcting controllers. We show that decomposition via partitions followed by the generation of synchronisation conditions and the removal of unnecessary transitions yields the desired result.

Let $\mathcal{A}$ be decomposed via partitions into $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\mathcal{A} \leq (\mathcal{A}_1 \parallel \mathcal{A}_2)$. Also assume that the covering map from the states of $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ to $\mathcal{A}$ is given by η. The initial state and the set of final states in $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ are precisely those which are mapped under η to the initial state or final states in the original automaton.

We now describe the technique of extracting the distributed alphabet, the component automata and the synchronising conditions. The first step is to synthesise the synchronisation restrictions and the second step is to obtain the distributed alphabet. To compute the synchronisation restrictions, for every action a, define $synch_a$ to be $\{q \in (Q_1 \times Q_2) \mid \exists q' \in Q, \eta(q) \xrightarrow{a} q'\}$. Let $synch$ represent the collection the synchronisation sets. The following proposition characterises the importance of $synch$.

Lemma 4. *Let $\mathcal{A}$ be decomposed via partitions into $\mathcal{A}_1$, $\mathcal{A}_2$. Also assume that synch as specified above has been obtained.*

If the state (q_1,q_2) of the automaton $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ is reachable under synch, $\eta((q_1,q_2))$ is defined and is identical to the intersection of q_1 and q_2.

Proof: Initially this is true as the initial state is (q_1,q_2) where $q_0 \in (q_1 \cap q_2)$. If $(q_1,q_2) \xrightarrow{a} (q'_1,q'_2)$, it is essential that $q_1 \xrightarrow{a}_1 q'_1$ and $q_2 \xrightarrow{a}_2 q'_2$. The definition of *synch* requires $q_1 \cap q_2 \xrightarrow{a} s$. But then s has to belong to q'_1 and q'_2. As the partitions are orthogonal, $q'_1 \cap q'_2$ is identical to $\{s\}$. Therefore, $\eta((q'_1,q'_2))$ will be s. •

3.1 Distributed Alphabet

Thus far we have only generated the two automata along with the synchronisation restrictions. However, the alphabets of the two automata are identical. We now describe the procedure to drop certain symbols (along with their transitions) from the component automata thereby obtaining truly distributed automata.

The distributed alphabet is obtained by deleting all "redundant" symbols and hence the associated transitions. The idea behind removing redundant symbols is that any symbol that cannot change the local state of machine (say $\mathcal{A}_2$) or block the other component (say $\mathcal{A}_1$), can be ignored in $\mathcal{A}_2$. Furthermore, all transitions on the symbol must also be permitted by *synch*. In other words, every a a-move of the the first component is independent of the second component and is permitted by the synchronisation requirements.

More precisely, a symbol of $\mathcal{A}_2$ is *redundant* if the following two conditions hold.

1. $\mathcal{A}_2$ has only self-loops on the symbol a. That is, for every q_2 belonging to Q_2, $q_2 \xrightarrow{a}_2 q_2$.
2. $\forall q_1 \in Q_1$, ($q_1 \xrightarrow{a}_1$ implies $\forall q_2 \in Q_2$, $(q_1,q_2) \in synch_a$).

A symmetric definition for the redundant symbols of $\mathcal{A}_1$ will be assumed.

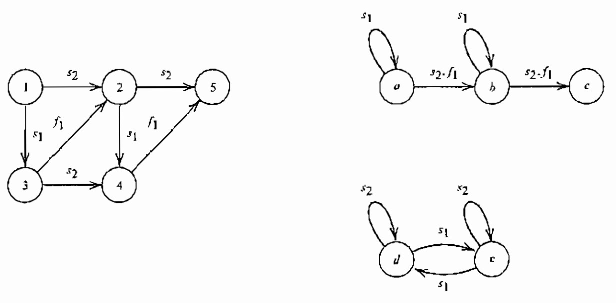

Fig. 2. Example

Consider the automaton and its decomposition shown in figure 2 which will illustrate the identification of a redundant symbol. Let a denote the state $\{1,3\}$, b denote the state $\{2,4\}$, c denote the state $\{5\}$, d denote the state $\{1,2,5\}$ and e denote the state $\{3,4\}$. The covering map η is given as follows:

$\eta(a,d) = 1, \eta(a,e) = 3, \eta(b,d) = 2, \eta(b,e) = 4, \eta(c,d) = 5$ and $\eta(c,e)$ is not defined.

The set $synch_{s_2}$ is $\{(a,d),(a,e),(b,d)\}$. The state (b,e) does not belong to this set as there is no s_2 transition from state 4. Hence the self-loops on s_2 in the second component cannot be removed. However, if *synch* permitted all moves, the self-loops can be removed. A more detailed case is presented in section 4.

Let $\mathcal{A}$ over Σ be decomposed via partitions into $\mathcal{A}_1$ and $\mathcal{A}_2$. Also assume that the synchronisation sets *synch* has been synthesised. Let Σ_i' be the subset of Σ obtained by eliminating redundant symbols with $\mathcal{A}_i'$ to be the restriction of $\mathcal{A}_i$ to Σ_i'. Given $\mathcal{A}_1'$ over Σ_1' and $\mathcal{A}_2'$ over Σ_2' the synchronisation constraints are now be restricted to only those actions that occur in both automata.

Lemma 5. *Under the above conditions, if* $\Sigma_1' \cup \Sigma_2' = \Sigma$, *the language accepted by* $\mathcal{A}$ *(say L) is identical to the language accepted by restricted product automaton derived from* $\mathcal{A}_1', \mathcal{A}_2'$ *and synch (say M).*

Proof: We first show that $L \subseteq M$. The complete proof can be found in [Hol82]. Here we present the general idea. Let $a_0a_1 \cdots a_n$ belong to L. Hence there is a run (a sequence of states) of $\mathcal{A}$ $q_0, q_1, \ldots q_{n+1}$ such that q_0 is the initial state and q_{n+1} belongs to F. Also $q_i \xrightarrow{a_i} q_{i+1}$. As η is a covering map, there has to exist a global state (q_1^i, q_2^i) mapped onto to q_i. Similarly a global state (q_1^{i+1}, q_2^{i+1}) has to exist with the proviso that (q_1^i, q_2^i) has an a transition to (q_1^{i+1}, q_2^{i+1}). The initiality and finality conditions follow directly from the definition.

We now show $M \subseteq L$. Let $a_0a_1 \cdots a_n$ belong to M. Hence there is a global run (q_1^0, q_2^0), (q_1^1, q_2^1), $\ldots$, (q_1^{n+1}, q_2^{n+1}). We translate this run into a run of the original automaton. For each (q_1^i, q_2^i), there will be a state q_i such that $\eta(q_1^i, q_2^i) = q_i$. This is guaranteed by Lemma 4.

Consider the transition $(q_1^i, q_2^i) \xrightarrow{a_i} (q_1^{i+1}, q_2^{i+1})$ made by the restricted product automaton. In this case *synch* will contain (q_1^i, q_2^i). If a_i belongs to both Σ_1' and Σ_2', the transition will be permitted in the original automaton. Otherwise, without loss of generality assume that a_i does not belong to Σ_1'. In that case a_i is redundant in $\mathcal{A}_1$ which means that in the decomposed automaton $q_1^i \xrightarrow{a_i} q_1^i$. Once again this implies that the transition will be permitted in the original automaton.

In otherwords, if the global automaton had an a_i transition in state (q_1^i, q_2^i), $synch_a$ would have to contain (q_1^i, q_2^i). This is possible only if q_i had an a_i transition. ●

The above proposition is valid for any $\mathcal{A}$, $\mathcal{A}_1'$ and $\mathcal{A}_2'$ with the necessary covering map and synchronisation sets. A few simplifications can be made for $\mathcal{A}_1'$ and $\mathcal{A}_2'$ that are derived by using partitions for the decomposition process, i.e., with subsets of states as the state space. This is related to proposition 4. From the construction of the decompositions, each state of $\mathcal{A}_1$ and $\mathcal{A}_2$ is a subset of the states of $\mathcal{A}$. As the partitions are orthogonal, η corresponds precisely to the intersection of the appropriate sets. Furthermore, if the intersection of two sets is empty, η on those states will be undefined. By proposition 4 it follows that such a global state will not be reachable. We reiterate that this observation would not hold without *synch*.

Furthermore, the transition relation on a given input symbol for $\mathcal{A}_1$ (similarly for $\mathcal{A}_2$) is defined to the union of the transitions on the same symbol for the original automaton. Hence for action a which belongs to only to one automaton, the set $synch_a$ can be ignored. The details of this construction follows from the proof in [Hol82, page 80].

Sometimes it is helpful to add 'irrelevant' self loops so that a symbol can be abstracted away to obtain automata that are more loosely coupled. Towards, this the following observation will help.

Let $q_2 \in \mathcal{A}_2$ such that q_2 has no a move. Furthermore, for every $q_1 \in \mathcal{A}_1$ such that q_1 has an a move, let $q_1 \cap q_2$ be the emptyset. Then adding a self loop on a to q_2 does not change the language accepted by the automata in question. This is because the global state (q_1, q_2) is not reachable and hence the question of synchronisation does not arise. This concludes the discussion of generating synchronising automata.

We now show that the two component automata are non-conflicting. Ensuring this property is key to our approach. Towards that we assume the following (a property called *trim*) about the original automaton. First, we assume that it has no unreachable states. Second, we also assume that from every state, a final state is reachable. The conversion of an automaton to an equivalent trim automaton is standard [HU79].

Lemma 6. Let $\mathcal{A}$ be a trim deterministic finite automaton decomposed (by partitions) into $\mathcal{A}_1$ and $\mathcal{A}_2$ with *synch*. Let L_1 be the language accepted by the automaton $\mathcal{A}_1$ and L_2 be the language accepted by the automaton $\mathcal{A}_2$. The joint behaviour of $\mathcal{A}_1$ and $\mathcal{A}_2$ under *synch* is non-conflicting.

Proof: Let $\alpha \in (\widetilde{L_1} \cap \widetilde{L_2})$. As the state space of the automata accepting $\widetilde{L_i}$ is identical to Q_i, we assume that the joint behaviour of the two components on α leads to the global state (q_1, q_2). Now we have to show that $\alpha\beta$ belongs to both L_1 and L_2 for some $\beta \in \Sigma^*$.

The original automaton on reading α would have reached a state q such that $\eta(q_1, q_2)$ equals q. If (q_1, q_2) does not have any move it implies that q does not have any move. But the original automaton was trim. Hence q has to be the final state in which case (q_1, q_2) is a final state. Hence β being the empty string suffices.

Otherwise, q will accept some β towards a final state. As the languages accepted by the two automata are the same, the joint behaviour should be able to exhibit β to a final state. This shows that the two automata are non-conflicting. ●

To summarise, we have shown that a decomposition based on partitions yields non-conflicting Zielonka asynchronous automata.

A prototype program to help the user construct orthogonal partitions has been written. The program written in Hugs (a variant of Haskell) [Jon94a] works on an explicit state transition representation. A program to handle symbolic representation is under construction. Given a transition system, the user can specify an initial seed partition. There are two ways of specifying this. The first is a direct enumeration of the seed partition. The second technique is by using a list of actions. This list represents part of the desired distributed alphabet for the resulting controller. This list of actions can be automatically generated using the original distributed alphabet. But the user can also provide any set of actions. A user defined set of actions is useful in the context of PLCs where inputs/outputs are explicitly specified. In this case, a seed partition is constructed such that for any action in the given list all transitions are self-loops. For example, if a

belongs to the list of actions and if $q \xrightarrow{a} q'$, the states q and q' are identified. The program then computes the finest admissible partition consistent with the seed partition. Further refinement of the partition is possible by invoking special functions. Given a list of seed partitions the program determines if the resulting admissible partitions are orthogonal. If they are orthogonal, it computes the *synch* sets and removes the redundant symbols.

4 Examples

We present two examples to illustrate the approach described earlier. The first is an asynchronous automaton that cannot be represented by the product of two automata over a given alphabet. The second is a larger example from the theory of discrete event systems.

Consider the global automaton accepting the language $((ab+ba+aabb+abab+abba+baba+bbaa+baab)d)^*$. From [Zie87] it is known that no product decomposition over the distributed alphabet $(\{b,d\},\{a,d\})$ can exist. Therefore, in order to obtain a correct decomposition global information in the form of synchronisation restrictions is required. Applying the decomposition yields the automata shown in figure 3. That is the first partition is $\{\ \{1,2,4\}, \{3,5,7\}, \{6,8,9\}\}$ while the second partition is $\{\ \{1,3,6\}, \{2,5,8\}, \{4,7,9\}\ \}$. The self loops on a (in the first component) and b (in the second component) do not have any synchronisation restrictions. So they satisfy the requirements for being redundant and hence can be removed. This results in $\{\ \{b,d\}, \{a,d\}\ \}$ as the distributed alphabet.

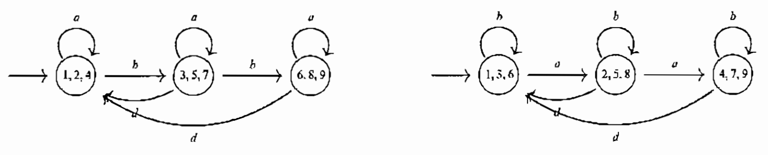

Fig. 3. The Component Automata

The covering map η can be described by the usual intersection of the partitions. From η one can compute the set $synch_d$ to contain precisely two elements, viz., $(\{3,5,7\},\{2,5,8\})$ and $(\{6,8,9\},\{4,7,9\})$. This is because a d transition is possible only from states 5 and 9. Another way to look at this is that the global state $(\{3,5,7\},\{4,7,9\})$ cannot exhibit a d. Furthermore, this state could be reachable as their intersection is non-empty (i.e., 7). But there is no d transition in state 7 in the original automaton and hence this transition needs to be disabled. By a similar argument, the d transition in the global state $(\{6,8,9\},\{2,5,8\})$ needs to be disabled as state 8 has no d transition.

Our next example is based on the controller for a transfer line containing two machines and a tester linked by buffers as shown below.

$$\begin{tikzcd} M_1 \arrow[r] & B_1 \arrow[r] & M_2 \arrow[r] & B_2 \arrow[r] & T \arrow[lll, bend left] \end{tikzcd}$$

This example has been discussed in [Won04]. This example illustrates both the strength and weakness of the decomposition approach.

The machine M_1 can start (s_1) or finish (f_1). When the machine finishes it places an item in the buffer B_1. If B_1 contains any element the machine M_2 can start (s_2) and when it finishes (f_2) it places an item in B_2. The tester can pick up an item from B_2 (s_3) and either accept it (a) or reject it (r) in which case it places it in buffer B_1. The starting of the machines is controllable and the key is to avoid buffer overflow or underflow. We will assume that buffer B_1 has capacity three and buffer B_2 has capacity one.

While we present the behaviour of the controller and the decomposition using explicit state transition diagrams, symbolic synthesis techniques [BHG$^+$93,NW94,MW98] are actually used in practice. However, it is hard to discuss the behaviour of such controller using symbolic representations such as BDDs [Bry86]. While the state transition diagrams may look complex, they illustrate the available symmetry which is actually exploited in the decomposition process. Issues related to the realisation of synthesised controllers are discussed in [Zha96].

The monolithic controller for this system can be synthesised and is partially shown in figure 4. The transitions on a and r have been omitted in the diagram. All other transitions are shown. There are 22 more transitions (for example from state 9 to state 10 on an a, from state 9 to state 15 on an r). The purpose of this figure is only to illustrate the regular structure present in the controller. Also, the controller synthesised using symbolic representations cannot be easily understood. In general the action a signifies that that an item has left the system. Hence it allows M_1 to start and hence introduce a new item in the system while an r action is as if an old item is reintroduced into the system. To begin with the controller can allow three items to be introduced into the system ($s_1, f_1, s_1, f_1, s_1, f_1$ leading to state 15). After that an s_1 can only occur after an a occurs.

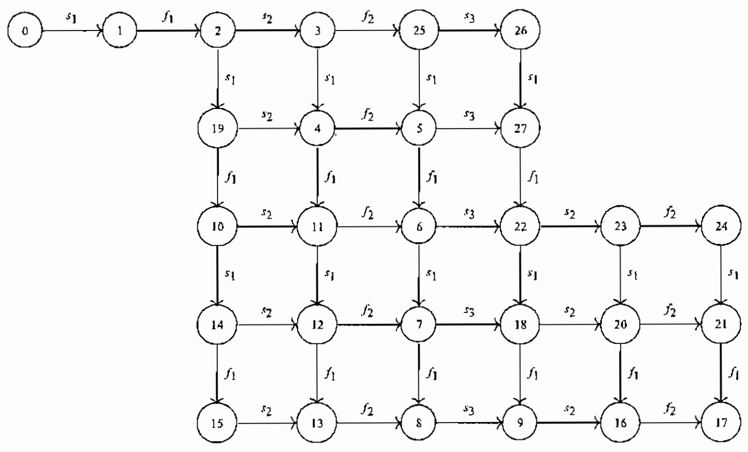

Fig. 4. Controller

By applying the theory of partitions twice, we obtain the three component automata shown in figures 5, and 6. This was achieved by the user guiding the prototype program. The first partition was achieved by requiring self-loops on the actions s_1, f_1, a and r. This is because these actions have no effect on B_2. The second partition is obtained by requiring self-loops on s_1 and f_2 (trying to obtain the controller that prevents underflow of B_1) while the third partition was obtained using self-loops on the actions s_2, f_2, s_3 (trying to prevent the overflow of B_1).

The *synch* set was initially calculated. It contained elements for the actions a and r. For example, the decomposed global state (Q_1, R_2, S_3) (corresponding to state 2) will not be present in $synch_a$. This is because the decomposed global state can exhibit an a but an a action is not possible from state 2. For the sake of simplicity we do not present the complete *synch* set. By applying the simplifications discussed in the next section it can be shown that no extra global information is required. As an aside, the *synch* set can be represented as a constraint on the variables shared by the two component automata.

In the diagrams the state Q_1 refers to the set $\{0 \ldots 4{,}9 \ldots 16{,}18 \ldots 20{,}22{,}23{,}26{,}27\}$, Q_2 to the set $\{5 \ldots 8{,}17{,}21{,}24{,}25\}$, R_1 to the set $\{0{,}1\}$ R_2 to the set $\{2{,}6{,}7{,}11{,}12{,}16{,}17{,}19\}$ R_3 to the set $\{3 \ldots 5{,}20{,}21{,}23{,}\ldots{,}25\}$, R_4 to the set $\{8{,}10{,}13{,}14\}$, R_5 to the set $\{18{,}22\}$, R_6 to the set $\{26{,}27\}$, R_7 to the set $\{9\}$, R_8 to the set $\{15\}$, S_1 to the set $\{0\}$, S_2 to the set $\{1\}$, S_3 to the set $\{2{,}3{,}25{,}26\}$, S_4 to the set $\{4{,}5{,}19{,}27\}$, S_5 to the set $\{6{,}10{,}11{,}22{,}\ldots{,}24\}$, S_6 to the set $\{7{,}12{,}14{,}18{,}20{,}21\}$ and S_7 to the set $\{8{,}9{,}13{,}15{,}\ldots{,}17\}$.

The automata in figure 5 ensure that buffer B_2 never overflows or underflows and that B_1 avoids underflow with respect to machine M_2. For example, an action f_1 or an action r inserts an element into B_1 after which M_2 can be started. The action s_2 removes an element from the buffer and when the count reaches 0, the machine M_2 cannot be started. This ensures that the buffer B_1 cannot underflow. However, the various transitions on the actions s_3, a and r are necessary to maintain orthogonality of the projections. We return to this automata later. The automaton in figure 6 specifies the control on buffer B_1 with respect to M_1 and more or less avoids overflow. Machine M_1 has to know the number of parts in the system before it can decide to produce more parts. This is because the tester could reject all the parts in the system and a safe behaviour requires that B_1 has at least that many free places. For the sake of readability the state associations (derived from the original automaton) are not shown in the decomposed automata.

The procedure to obtain non-conflicting controllers can be summarised as follows:

1. Synthesise the single controller.
2. Obtain two orthogonal partitions along with the synchronisation restrictions.
3. Remove redundant symbols and thereby obtain a distributed alphabet.

The current theory focuses only on partitioning into two components. The direct decomposition into multiple components needs further research.

The limitation of the partition approach is clearly demonstrated in the second automaton in figure 5 and figure 6 In the two automata shown, certain states deserve special attention. We focus our discussion on the state marked R_1. Similar arguments

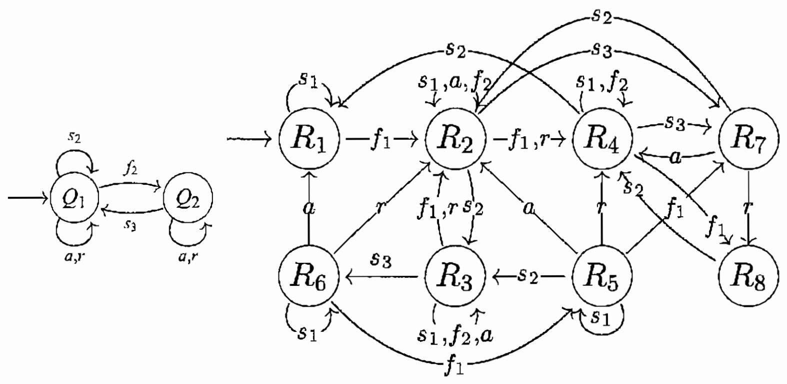

Fig. 5. *Buffer*$_2$ and *Buffer*$_1$ Underflow Control

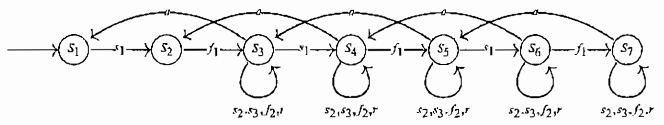

Fig. 6. *Buffer*$_1$:Overflow Control

hold for the state marked R_6, R_7, R_8, S_1 and S_2. The decomposition process does not yield any self loop on the actions a and f_2 for the state R_1. Hence the actions s_1, a and f are not redundant. This forces a tighter coupling than desired; but as we show in the next section this tighter coupling can be loosened.

Also, for this particular problem the states R_1, R_3 and R_6 can be merged. This would then introduce self loops on various actions to the automaton. This could then be used to obtain a more loosely coupled automaton. But the merging of R_1, R_3 and R_6 does not yield orthogonal partitions. Similarly, states R_5 and R_2 can be merged and R_7 and R_4 can be merged. This can be done for this problem as state R_1, R_3 and R_6 effectively remember the buffer B_1 being empty, R_2 and R_5 indicate the buffer containing one item, R_4 and R_7 the buffer containing two items.

However, such analyses are problem specific and beyond what the theory of partitions yields. That is, the partition technique has to be augmented with problem specific analysis to further simplify the controllers. This issue is considered in the following section.

5 Including the Plant Model

So far we have not used the fact that the original automaton was a controller for a given plant. We have also not used the division of the input alphabet into controllable and uncontrollable actions. Recall that only controllable actions can be disabled. This implies that if a controller cannot exhibit an uncontrollable action in a given state, the environment cannot exhibit the action. Therefore, augmenting the controller with a transition on an uncontrollable action does not change the overall behaviour. Similarly, adding a transition on a controllable action where the plant cannot use it does not enable any new behaviours.

The general observation (for product automata) is as follows. Let q_1 be a state in the first component and q_2 be a state in the second component. If both the states have no a transition from them adding an a transition to *only one* of them will not affect their joint behaviour. As we are synthesising the controllers, we can add extra transitions to them without enabling unspecified behaviours.

By using transitions that the plant cannot perform, the component controllers automata obtained by the partition technique can be simplified by reducing the required global information. This simplification will alter the language accepted by the controller but will not affect the joint behaviour of the plant and the controller.

We now present this more precisely. Let $\mathcal{A}_C$ represent the controller and $\mathcal{A}_P$ represent the plant. The effect of the controller on the plant can be characterised as follows. Let $cp : Q^C \longrightarrow 2^{Q^P}$ such that $q_p \in cp(q_c)$ iff the state (q_p, q_c) is reachable in $(\mathcal{A}_P \parallel \mathcal{A}_C)$. In other words, $cp(q_c)$ identifies all the states the plant could be in while the controller is in state q_c.

If for every q_p belonging to $cp(q_c)$, q_p has no a-move (a can be any action) and q_c has no a transition, consider $\mathcal{A}_C$ augmented with the transition $q_c \xrightarrow{a} q_c$. It is easy to see that the joint behaviour of the plant and the augmented automaton is identical to the joint behaviour of the plant and the original automaton.

We now focus on the component controller automata. Let $\mathcal{A}_P$, $\mathcal{A}_C$ and cp be as before. Let $\mathcal{A}_{C_1}$ and $\mathcal{A}_{C_2}$ be the decomposition of $\mathcal{A}_C$ with *synch* the synchronisation set.

Consider an action a such that all a transitions in $\mathcal{A}_{C_1}$ are only self-loops. Let $N_a = \{\, q \in \mathcal{A}_{C_1} \mid q \not\xrightarrow{a}_1 \,\}$. The set N_a identifies the states that have no a transitions. Let q_1 be a state belonging to N_a. If for every state (q_2) of the automaton $\mathcal{A}_{C_2}, q_2 \xrightarrow{a}_2$ implies $cp(\eta(q_1, q_2)) \not\xrightarrow{a}_P$, augment $\mathcal{A}_{C_1}$ with the transition $q_1 \xrightarrow{a}_1 q_1$ and let *synch'* be *synch* augmented with $\{\, (q_1, q_2) \mid q_2 \xrightarrow{a}_2 \,\}$. Call this new automaton $\mathcal{A}'_{C_1}$. By a symmetric process obtain $\mathcal{A}'_{C_2}$ from $\mathcal{A}_{C_2}$.

Under the above assumptions the following property is valid. The proof of the above proposition is obvious and follows from the definition of the product of automata.

Lemma 7. *The joint behaviour of $\mathcal{A}'_{C_1}$ and $\mathcal{A}'_{C_2}$ under synch' with $\mathcal{A}_P$ is identical to the joint behaviour of $\mathcal{A}_{C_1}$ and $\mathcal{A}_{C_2}$ under synch with $\mathcal{A}_P$.*

We conclude this section with a discussion of an example from the previous section. The first automaton shown in figure 5 (the controller for the second buffer) has self-loops on the actions a and r which are uncontrollable actions. By considering the other decomposed automata *synch*$_a$ and *synch*$_r$ can be augmented with the extra states such that actions a and r become redundant. Hence the controller for that buffer has only s_2, f_2 and s_3 in its alphabet. The new controller is

$$\begin{tikzcd} \arrow[r] & Q_1 \arrow[loop above, "s_2"] \arrow[r, bend left, "f_2"] & Q_2 \arrow[l, bend left, "s_3"] \end{tikzcd}$$

It shows clearly that the controller is unconcerned about the outcome of the testing process as this affects only the first buffer.

Consider the automaton shown in figure 6. When the automaton is in state S_1 or S_2 (the state of the other component automata being immaterial) the plant cannot exhibit an f_2 or an r. This is because both buffers are empty. Hence one can add self loops on f_2 and r without changing the joint behaviour. Now by applying the definition of redundancy, the symbols f_2 and r can be removed from the automaton; thus obtaining a simpler automaton. This is because the controller has to be prepared for an item to be rejected (an uncontrollable action) and hence reserves a slot for it. However, if the item is accepted the slot can be released.

Note that self-loops for the symbols s_2 and s_3 cannot be added as the plant can indeed perform these actions and the controller explicitly disables them. Although one could add a-loops to the state S_1 or S_2, these are not useful as the other states do not have self-loops on a.

The simplifications can be summarised as follows. In the first case we only augment the *synch* sets while in the second case we add extra transitions as well as augment the *synch* sets. In both cases the desired outcome is identical, viz., to make a particular action redundant. As these changes are purely on the structure of the automata they can be fully automated.

6 Conclusion and Future Work

We have presented a framework in which a single controller can be decomposed into non-conflicting controllers. Although we have taken a distributed system, constructed a monolithic system and then synthesised a controller, we have suggested in the introduction that this be used only when the controllers generated from the distributed system are conflcting.

We now present an example to show the limitations for the partition based technique. Consider the automaton

$$\begin{tikzcd}
1 \arrow[loop above, "s_1"] \arrow[r, bend left, "r_2"] \arrow[rr, bend right, "f_1"'] & 0 \arrow[loop above, "s_1{,}r_1"] \arrow[l, bend left, "b_2"] \arrow[r, bend left, "f_1"] & 2 \arrow[loop above, "b_2{,}r_2"] \arrow[l, bend left, "s_2"]
\end{tikzcd}$$

(with 0 as the start state).

For the first automaton the partition $\{\{0,1\},\{2\}\}$ suffices. However, equating states 0 and 2 or 1 and 2 results in the need to equate all three states. Hence two orthogonal partitions are not possible.

A method based on set systems [HS66] (instead of partitions arbitrary subsets are used to obtain the components) could be developed. For example, the set system $\{\{0,2\},\{0,1\}\}$ along with the partition $\{\{0,1\},\{2\}\}$ yields a decomposition. But the drawback of this approach is that one may need to consider non-deterministic systems which then have to be determinised. At this point it is not clear if the method based on set systems is as clean as the one based on partitions for deterministic systems.

Acknowledgments

The author thanks the synthesis and modular verification group at Siemens research for their support. Special thanks to Burkhard Stubert, Klaus Winkelmann and Jorge Cuellar for their detailed comments. One of the anonymous referee deserves special thanks for providing insightful comments on the original submission.

References

[BHG$^+$93] S. Balemi, G. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. Franklin. Supervisory Control of a Rapid Thermal Multiprocessor. *IEEE Trans. on Automatic Control*, 38:1040–1059, 1993.

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35(8):677–691, 1986.

[dRLP98] W-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Different: COMPOS 1997*, volume LNCS 1536, Bad Malente, Germany, 1998.

[Hol82] W. M. L. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1982.

[HS66] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice Hall, 1966.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[Jon94a] M. Jones. *An introduction to HUGS, Version 1.01*. University of Nottingham, 1994.

[Jon94b] B. Jonsson. Compositional specifi cation and verifi cation of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.

[MW98] H. Melcher and K. Winkelmann. Controller synthesis for the production cell case study. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 24–33, New York, March 4–5 1998. ACM Press.

[NW94] K. Noekel and K. Winkelmann. CSL. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems: Case Study Production Cell*, volume LNCS 891, pages 55–74. Springer Verlag, 1994.

[PTV01] A. Puri, S. Tripakis, and P. Varaiya. Problems and examples of decentralized observation and control for discrete event systems. In *Symposium on the Supervisory Control of Discrete Event Systems*, 2001.

[Ram95] R. Ramanujam. A local presentation of synchronizing systems. In *Structures in Concurrency Theory*, Springer Workshops in Computing, pages 91–118. Springer Verlag, 1995.

[RW89] P. J. G. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[SEM03] A. Stefanescu, J. Esparza, and A. Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *Proc. of CONCUR'03*, LNCS 2761, pages 27–41, 2003.

[SW04] R. Su and W. M. Wonham. Supervisory Reduction for Discrete-Event Systems. *Discrete Event Dynamic Systems*, 14(1):31–53, January 2004.

[Tri04] S. Tripakis. Undecidable problems of decentralized observation and control on regular languages. *Information Processing Letters*, 90(1):21–28, 2004.

[Won04] W. M. Wonham. Supervisory control of discrete-event systems (updated notes). Technical report, Systems Control Group, University of Toronto, Canada, 2004.

[WR88] W. M. Wonham and P. J. G. Ramadge. Modular Supervisory Control of Discrete Event Systems. *Mathematics of Control, Signals and Systems*, 1(1):13–30, 1988.

[Zha96] Y. Zhang. Software for State-Event Observation Theory and its Application to Supervisory Control. Master's thesis, Department of Control Engineering, University of Toronto, Canada, 1996.

[Zie87] W. Zielonka. Notes on Finite Asynchronous Automata. *RAIRO: Theoretical Informatics and Applications*, 21(2):101–135, 1987.