Synthesis of Distributed Algorithms Using Asynchronous Automata

Alin Ştefănescu[1], Javier Esparza[1], and Anca Muscholl[2]

[1] Institut für Formale Methoden der Informatik
Universitätsstr. 38, 70569 Stuttgart, Germany
[2] LIAFA, Université Paris VII
2, place Jussieu, case 7014, F-75251 Paris Cedex 05

Abstract. We apply the theory of asynchronous automata to the synthesis problem of closed distributed systems. We use safe asynchronous automata as implementation model, and characterise the languages they accept. We analyze the complexity of the synthesis problem in our framework. Theorems by Zielonka and Morin are then used to develop and implement a synthesis algorithm. Finally, we apply the developed algorithms to the classic problem of mutual exclusion.

1 Introduction

We address the problem of automatically synthesising a finite-state, closed distributed system from a given specification. Seminal papers in this area are [EC82, MW84], where synthesis algorithms from temporal logic specifications are developed. The algorithms are based on tableau procedures for the satisfiability problem of CTL and LTL.

These approaches suffer from the limitation that the synthesis algorithms produce a sequential process P, and not a distributed implementation, i.e., a tuple $(P_1, \ldots, P_n)$ of communicating processes. The solution suggested in these works is to first synthesise the sequential solution, and then decompose it. However, since distribution aspects like concurrency and independency of events are not part of the CTL or LTL specification (and cannot be, since they are not bisimulation invariant), the solution may be impossible to distribute while keeping the intended concurrency. (This is in fact what happens with the solutions of [EC82,MW84] to the mutual exclusion problem)

A better approach to the problem consists in formally specifying not only the properties the system should satisfy, but also its architecture (how many components, and how they communicate). This approach was studied in [PR89] for *open* systems, in which the environment is an *adversary* of the system components, and the question is whether the system has a strategy that guarantees the specification against all possible behaviours of the environment. The realization problem (given the properties and the architecture, decide if there exists an implementation) was shown to be undecidable for arbitrary architectures, and decidable but non-elementary for hierarchical architectures vs. LTL specifications. Recent work [KV01] extends the decidability result (and the upper bound)

R. Amadio, D. Lugiez (Eds.): CONCUR 2003, LNCS 2761, pp. 27–41, 2003.

to CTL* specifications and linear architectures. To the best of our knowledge the synthesis procedures have not been implemented or tested on small examples.

In this paper we study the realization problem for the simpler case of *closed* systems, the original class of systems considered in [EC82,MW84]. This problem has been studied with unlabelled Petri nets (see e.g. [BD98]) and product transition systems (see [CMT99] and the references therein) as notions of implementation. In this paper, we attack the problem using *asynchronous automata* [Zie87,DR95]. Asynchronous automata can be seen as a tuple of concurrent processes communicating in a certain way (or as 1-safe *labelled* Petri nets). In our approach, a specification consists of two parts: a regular language L over an alphabet Σ of actions, containing all the finite executions that the synthesised system should be able to execute, and a tuple $(\Sigma_1, \ldots, \Sigma_n)$ of *local alphabets* indicating the actions in which the processes to be synthesised are involved; an action can be executed only if all processes involved in it are willing to execute it. The synthesis problem consists of producing a so-called *safe* asynchronous automaton whose associated processes have $(\Sigma_1, \ldots, \Sigma_n)$ as alphabets, and whose language is included in L (together with some other conditions to remove trivial solutions). The main advantage of our approach with respect to those of [BD98, CMT99] is its generality: Unlabelled Petri nets and product transition systems can be seen as strict subclasses of safe asynchronous automata.

The first two contributions of the paper are of theoretical nature. The first one is a refinement of Zielonka's theorem [Zie87], a celebrated central result of the theory of Mazurkiewicz traces. The refinement characterises the languages recognised by safe asynchronous automata, which we call *implementable* languages. (This result was also announced in [Muk02] without proof.) This result allows to divide the synthesis problem into two parts: (1) given a specification L, $(\Sigma_1, \ldots, \Sigma_n)$, decide if there exists an implementable language $L' \subseteq L$, and (2) given a such L', obtain a safe asynchronous automaton with L' as language. In the second contribution, we find that part (1) is undecidable, therefore we restrict our attention to an NP-complete subclass of solutions for which reasonable heuristics can be developed.

The third and main contribution of the paper is the development of heuristics to solve (1) and (2) in practice, their application to the mutual exclusion problem, and the evaluation of the results. The heuristic for (2) uses a result by Morin [Mor98] to speed up the synthesis procedure given by Zielonka in [Zie87]. Our heuristics synthesise two (maybe not 'elegant' but) new and far more realistic shared-variables solutions to the mutex problem than those of [EC82,MW84] (the results of [PR89], being for open systems and very generally applicable, did not provide any better automatically generated solution to the mutual exclusion problem). We make good use of the larger expressivity of asynchronous automata compared to unlabelled Petri nets and product transition systems: The first solution cannot be synthesised using Petri nets, and the second – the most realistic – cannot be synthesised using Petri nets or product transition systems.

The paper is structured as follows. Section 2 introduces asynchronous automata and Zielonka's theorem. In Sect. 3 we present the characterisation of implementable languages. Section 4 describes the synthesis problem together

with heuristics for the construction of a solution and discusses complexity issues. Section 5 shows the synthesis procedure at work on the mutual exclusion problem. All the proofs can be found in [SEM03].

2 Preliminaries

We start with some definitions and notations about automata and regular languages. A *finite automaton* is a five tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of *states*, Σ is a finite alphabet of *actions*, $I, F \subseteq Q$ are sets of *initial* and *final* states, respectively, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. We write $q \xrightarrow{a} q'$ to denote $(q, a, q') \in \delta$. The language recognised by $\mathcal{A}$ is defined as usual. A language is *regular* if it is recognised by some finite automaton. Given a language L, its *prefix closure* is the language containing all words of L together with all their prefixes. A language L is *prefix-closed* if it is equal to its prefix closure. Given two languages $L_1, L_2 \subseteq \Sigma^*$, we define their *shuffle* as $\text{shuffle}(L_1, L_2) := \{u_1 v_1 u_2 v_2 \dots u_k v_k \mid k \geq 1, u_1 \dots u_k \in L_1, v_1 \dots v_k \in L_2 \text{ and } u_i, v_i \in \Sigma^*\}$.

We recall that regular languages are closed under boolean operations, that the prefix-closure of a regular language is regular, and that the shuffle of two regular languages is regular.

2.1 Asynchronous Automata

Let Σ be a nonempty, finite alphabet of *actions*, and let *Proc* be a nonempty, finite set of *process labels*. A *distribution* of Σ over *Proc* is a function $\Delta\colon \mathit{Proc} \to 2^{\Sigma} \setminus \emptyset$. Intuitively, Δ assigns to each process the set of actions it is involved in, which are the actions that cannot be executed without the process participating in it. It is often more convenient to represent a distribution by the *domain* function $\mathit{dom} : \Sigma \to 2^{\mathit{Proc}} \setminus \emptyset$ that assigns to each action the processes that execute it. We call the pair (Σ, dom) a *distributed alphabet*. A distribution induces an *independence relation* $\|\colon \Sigma \times \Sigma$ as follows: $\forall a, b \in \Sigma :\ a \| b \Leftrightarrow \mathit{dom}(a) \cap \mathit{dom}(b) = \emptyset$. I.e., two actions are independent if no process is involved in both. The intuition is that independent actions may occur concurrently.

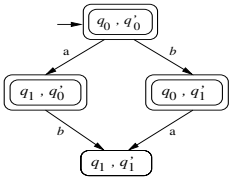
An asynchronous automaton over a distributed alphabet is a finite automaton that can be distributed into communicating local automata. The states of the automaton are tuples of local states of the local automata.

Definition 1. An *asynchronous automaton* $\mathcal{AA}$ over a distributed alphabet (Σ, dom) is a finite automaton $(Q, \Sigma, \delta, I, F)$ such that there exist

- a family of sets of *local states* $(Q_k)_{k \in \mathit{Proc}}$, and
- a relation $\delta_a \subseteq \prod_{k \in \mathit{dom}(a)} Q_k \times \prod_{k \in \mathit{dom}(a)} Q_k$ for each action $a \in \Sigma$

satisfying the following properties:

- $Q \subseteq \prod_{k \in \mathit{Proc}} Q_k$, with $I, F \subseteq Q$ initial and final states, and



$$\Sigma = \{a, b\}, \quad Proc = \{1, 2\}$$
$$dom(a) = \{1\}, \quad dom(b) = \{2\}$$
$$Q_1 = \{q_0, q_1\}, \quad Q_2 = \{q'_0, q'_1\}$$
$$\delta_a = \{(q_0, q_1)\}, \quad \delta_b = \{(q'_0, q'_1)\}$$
$$I = \{(q_0, q'_0)\}, F = \{(q_0, q'_0), (q_1, q'_0), (q_0, q'_1)\}$$

Fig. 1. An asynchronous automaton together with its formal description

– $(q, a, q') \in \delta \Leftrightarrow \begin{cases} \forall k \notin dom(a) : q_k = q'_k \\ \big((q_k)_{k \in dom(a)}, (q'_k)_{k \in dom(a)}\big) \in \delta_a \end{cases}$
where q_k denotes the k-th component of q, and $(q_k)_{k \in dom(a)}$ denotes the projection of q onto $dom(a)$.

The language recognised by an asynchronous automaton is the language it recognises as a finite automaton. If all δ_a's are functions and I contains only one element, then $\mathcal{AA}$ is called *deterministic*.

Figure 1 shows an asynchronous automaton. Intuitively, each set Q_k represents the set of states of a sequential component. Whether there is an a-transition between two states depends only on the projections of the states onto $dom(a)$, the local states of the other components are irrelevant; moreover the execution of a only changes the local state of the processes in $dom(a)$. In particular, if there is an a-transition between two global states q_1, q_2, then there must also be a-transitions between any states q'_1, q'_2 such that the projections of q_1, q'_1 and q_2, q'_2 on $dom(a)$ coincide. It is easy to see that, as a consequence, every asynchronous automaton satisfies the *independent* and *forward diamond rules*:

– **ID** : $q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_4 \wedge a \| b \Rightarrow \exists q_3 : q_1 \xrightarrow{b} q_3 \xrightarrow{a} q_4$
– **FD** : $q_1 \xrightarrow{a} q_2 \wedge q_1 \xrightarrow{b} q_3 \wedge a \| b \Rightarrow \exists q_4 : q_2 \xrightarrow{b} q_4 \wedge q_3 \xrightarrow{a} q_4$.

Finally, observe that the accepting conditions of asynchronous automata are *global*: We need to know the local states of *all* the processes in order to determine if the tuple of local states is a final state.

2.2 Zielonka's Theorem

Zielonka's theorem characterises the languages accepted by asynchronous automata. Given a distributed alphabet (Σ, dom), we say that $L \subseteq \Sigma^*$ is a *trace language* if L is closed under the independence relation $\|$ associated to dom:

$$\forall a, b \in \Sigma \text{ and } \forall w, w' \in \Sigma^* : \; wabw' \in L \wedge a \| b \Rightarrow wbaw' \in L.$$

Theorem 1. *[Zie87] Let (Σ, dom) be a distributed alphabet, and let $L \subseteq \Sigma^*$. L is recognised by a finite asynchronous automaton with distribution dom if and only if it is a regular trace language. Moreover, if L is recognised by an asynchronous automaton, then it is also recognised by a deterministic asynchronous automaton.*

The proof of the theorem is constructive. Zielonka defines an effectively computable equivalence relation $\approx_Z \subseteq \Sigma^* \times \Sigma^*$ of finite index. The definition of $\approx_Z$ can be found in [Zie87]. Now, let T_L be the infinite automaton having L as set of states, and $w \xrightarrow{a} wa$ as transitions. The asynchronous automaton of Theorem 1 is the quotient of T_L under $\approx_Z$. The size of the automaton is single exponential in the size of the minimal deterministic automaton recognising L, and double exponential in the size of *Proc*.

The following shows that in order to decide if a language is a regular trace language, it suffices to compute the minimal automaton recognising it, and check if it satisfies ID.

Proposition 1. *Let (Σ, dom) be a distributed alphabet, and let $L \subseteq \Sigma^*$ regular. The following conditions are equivalent:*

1. *L is a regular trace language;*
2. *the minimal deterministic finite automaton recognising L satisfies ID.*

3 Implementable Specifications

As mentioned in the introduction, we use regular languages as specification of the set of global *behaviours* of a distributed system, where a behaviour is a finite sequence of actions. In this setting, asynchronous automata are not a realistic implementation model. The reason is best explained by means of an example. Let $\Sigma = \{a, b\}$ and $dom(a) = \{1\}$, $dom(b) = \{2\}$, and consider the language $L = \{\epsilon, a, b\}$. Intuitively, (L, dom) cannot be implemented: Since L contains both a and b, and a and b are executed *independently* of each other, nothing can prevent an implementation from executing ab and ba as well, which do not belong to L. However, the asynchronous automaton of Fig. 1 recognises L. The reason is that we can choose the global final states as $\{(0,0),(1,0),(0,1)\}$, excluding $(1,1)$. (Notice that if we remove $(1,1)$ from the set of states the automaton is no longer asynchronous, because it does not satisfy FD.) In our context, in which runs of the automaton should represent behaviours of a distributed system, this is not acceptable: We cannot declare *a posteriori* that a sequence of actions we have observed is not a behaviour because the state it reaches as non-final.

This example shows that we have to restrict our attention to asynchronous automata in which all states reachable from the initial states are final. We call such automata *safe*[1].

As we mentioned in the introduction, the synthesis of closed distributed systems has been studied before using unlabelled Petri nets [BD98] and product transition systems [Mor98,CMT99] as implementation models. Both models can be seen as subclasses of safe asynchronous automata in which, for each action a, the relation δ_a satisfies an additional condition. In the case of Petri nets, δ_a

[1] Safe (asynchronous) automata were studied by Zielonka in [Zie89]. *Safe* there means something weaker: All reachable states are co-reachable (i.e. there is a path from that state to a final one). However, the difference between the two definitions of *safe* vanishes when the recognised language is prefix-closed.

must contain at most one element. In the case of product transition systems, δ_a must have a product form: There must be a family of relations $\delta_a^k \subseteq Q_k \times Q_k$ such that $\delta_a = \prod_{k \in Proc} \delta_a^k$.

In the rest of this section we obtain the equivalent of Theorem 1 and Proposition 1 for safe asynchronous automata.

Definition 2. A regular trace language $L \subseteq \Sigma^*$ is called *implementable* if it satisfies:

- *prefix-closedness*: $\forall w, w', w'' \in \Sigma^* : w = w'w'' \in L \Rightarrow w' \in L$
- *safe-branching*[2]: $\forall w \in \Sigma^* : wa \in L \wedge wb \in L \wedge a \| b \Rightarrow wab \in L$.

Theorem 2. *Let (Σ, dom) be a distributed alphabet, and let $L \subseteq \Sigma^*$. L is recognised by a finite safe asynchronous automaton with distribution dom if and only if it is an implementable trace language. Moreover, if L is recognised by a safe asynchronous automaton, then it is also recognised by a safe deterministic asynchronous automaton.*

A proof is given in [SEM03] and a constructive one follows from Proposition 3.

Proposition 2. *Let (Σ, dom) be a distributed alphabet, and let $L \subseteq \Sigma^*$ regular. The following conditions are equivalent:*

1. *L is an implementable language;*
2. *the minimal deterministic finite automaton recognising L is safe and satisfies ID and FD.*

This result provides an inexpensive test to check if a specification (L, dom) is implementable: Compute the minimal automaton recognising L and check if it satisfies ID and FD and if all its states are final. These checks have linear time complexity in the size of the minimal automaton and in the size of the independence relation generated by *dom*.

Remark 1. Testing whether a specification is implementable is PSPACE-complete, when the input is a regular expression or a non-deterministic automaton (it can be easily shown that both checking ID and FD are PSPACE-complete).

It is not difficult to show that implementable languages are a proper superset of the Petri net languages and the languages of product transition systems (see [Zie87]). As we will see in Sect. 5.1, this is the fact that will allow us to derive implementations in our case studies.

4 The Synthesis Problem

In our setting, the synthesis problem is: Given a distributed alphabet (Σ, dom) and a regular language L_{Spec}, represented as a deterministic finite automaton $\mathcal{A}_{Spec}$, is there a safe asynchronous automaton $\mathcal{AA}$ such that $L(\mathcal{AA}) \subseteq$

[2] In [Maz87] the property of *safe-branching* is called *properness*.

$L(\mathcal{A}_{Spec})$? In addition, we require that all the actions in Σ appear in $L(\mathcal{AA})$, because we are not interested in trivial solutions like $L(\mathcal{AA}) = \emptyset$ or partial solutions in which only some of the processes are executing actions. By definition, for a given language L, let $\Sigma(L) := \{a \in \Sigma \mid \exists u, v \in \Sigma^* \text{ with } uav \in L\}$ denote the actions appearing in L. Then, the set of actions *appearing* in an (asynchronous) automaton $\mathcal{A}$ is just $\Sigma(\mathcal{A}) := \Sigma(L(\mathcal{A}))$. We are now able to formulate:

Problem 1. (Synthesis problem) Given a distributed alphabet (Σ, dom) and a deterministic finite automaton $\mathcal{A}_{Spec}$ such that $\Sigma(\mathcal{A}_{Spec}) = \Sigma$, is there a safe asynchronous automaton $\mathcal{AA}$ such that $L(\mathcal{AA}) \subseteq L(\mathcal{A}_{Spec})$ and $\Sigma(\mathcal{AA}) = \Sigma$?

Theorem 3. *The synthesis problem is undecidable.*

Because of the undecidability of synthesis problem stated in Theorem 3, we attack a more modest but 'only' NP-complete problem, for which, as we can see, we can develop reasonable heuristics. This requires to introduce the notion of a subautomaton. We say that $\mathcal{A}' = (Q', \Sigma', \delta', I', F')$ is a *subautomaton* of $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ if $Q' \subseteq Q$, $\Sigma' \subseteq \Sigma$, $I' \subseteq I$, $F' \subseteq F$ and $\delta' \subseteq \delta$.

In the *subautomata synthesis problem* we search for the language of $\mathcal{AA}$ only among the languages of the subautomata of $\mathcal{A}_{Spec}$. More precisely, we examine the languages of the subautomata, which are obviously included in $L(\mathcal{A}_{Spec})$, and determine if some of them is the language of an asynchronous automaton. Since the languages of safe asynchronous automata are those implementable, what we in fact do is to consider the following problem:

Problem 2. (Subautomata synthesis) Given a distributed alphabet (Σ, dom) and a deterministic finite automaton $\mathcal{A}_{Spec}$ such that $\Sigma(\mathcal{A}_{Spec}) = \Sigma$, is there a safe subautomaton $\mathcal{A}'$ of $\mathcal{A}_{Spec}$ with $\Sigma(\mathcal{A}') = \Sigma$ satisfying ID and FD?

A positive solution to an instance of this problem implies a positive solution to the same instance of the synthesis problem.

Theorem 4. *The subautomata synthesis problem is NP-complete.*

Let us now summarize our approach:

1. Choose the set of actions Σ of the system and a distribution Δ.
2. Describe the 'good' behaviours of the system as a regular language L_{Spec}. Usually, we give L_{Spec} as a base language (e.g. a shuffle of local behaviours), from which we filter out undesired behaviours (e.g. behaviours that lead to two processes in a critical section).
3. Construct $\mathcal{A}$ (usually, the minimal deterministic finite automaton) satisfying $L(\mathcal{A}) = L_{Spec}$.
4. Find a safe subautomaton $\mathcal{A}'$ of $\mathcal{A}$ with $\Sigma(\mathcal{A}') = \Sigma$ satisfying ID and FD. (see Sect. 4.1)
5. Apply Theorem 2 to obtain a safe asynchronous automaton $\mathcal{AA}$ satisfying $L(\mathcal{AA}) = L(\mathcal{A}')$. [3] (see Sect. 4.2)

[3] Note that we can apply Theorem 2, because the language of a safe automaton satisfying ID and FD is implementable.

4.1 Constructing a Subautomaton

Given an automaton $\mathcal{A}$, finding a safe subautomaton $\mathcal{A}'$ satisfying $\Sigma(\mathcal{A}') = \Sigma$, ID, and FD is NP-complete, so in the worst case this is exponentially expensive. In our experiments, we found two natural heuristics helpful in this problem:

1. [*destructive*] Starting with the initial automaton $\mathcal{A}$, we remove states and transitions that prevent the properties of safety, ID and FD to hold. So, if we have non-final states, we remove them; if we have a conflict w.r.t. FD (e.g., $\exists q_1 \xrightarrow{a} q_2$ and $\exists q_1 \xrightarrow{b} q_3$ with $a \| b$, but there exists no state q_4 such that $\exists q_2 \xrightarrow{b} q_4$ and $\exists q_3 \xrightarrow{a} q_4$), we remove one of the transition involved in the conflict (e.g., removing $q_1 \xrightarrow{b} q_3$ will solve the conflict); something similar for ID. In the process of removal we want to preserve $\Sigma(\mathcal{A}') = \Sigma$.
2. [*constructive*] Starting with the empty subautomaton, we add states and transitions until we find a safe subautomaton $\mathcal{A}'$ satisfying ID and FD. We apply a breadth-first traversal together with a 'greedy strategy' which selects transitions labelled by new action names and we do not add transitions violating the ID and FD rules and we do not add non-final states.

In both of the above strategies, we stop when we find a subautomaton satisfying our properties. Therefore, in general, the first heuristic will produce a larger solution than the second one. Larger solutions represent more behaviours, so better implementations for our synthesis problem. Unfortunately, this large subautomaton will serve as an input for Zielonka's procedure and this may blow-up the state space of the solution. That is why the second heuristic is usually preferred and the experimental results in Sect. 5.2 witness this fact.

4.2 Constructing an Asynchronous Automaton

The proof of Zielonka's theorem provides an algorithm to automatically derive an asynchronous automaton from an implementable language L (obtained as in the previous subsection). We start by giving here a version of the algorithm. The version is tailored so that we can easily add a heuristic that we describe in the second half of the section. Loosely speaking, the algorithm proceeds by *unfolding the minimal deterministic automaton recognising L until an asynchronous automaton is obtained.*

Data structure The algorithm maintains a deterministic reachable automaton $\mathcal{A}$ in which all states are final. The transitions of $\mathcal{A}$ are coloured *green*, *red*, or *black*. The algorithm keeps the following invariants:

1. The automaton $\mathcal{A}$ is deterministic and recognises L.
2. *Green* transitions form a directed spanning-tree of $\mathcal{A}$, i.e., a directed tree with the initial state q_0 as root and containing all states of $\mathcal{A}$.
3. Let $W(q)$ be the unique word w such that there is a path $q_0 \xrightarrow{w} q$ in the spanning-tree. For any $q \neq q'$ we have $W(q) \not\approx_Z W(q')$. (Notice that if a transition $q \xrightarrow{a} q'$ is green, then $W(q) \cdot a = W(q')$.)
4. A transition $q \xrightarrow{a} q'$ is *red* if $W(q) \cdot a \not\approx_Z W(q')$.

5. All other transitions are *black*.

Initially, $\mathcal{A}$ is the minimal deterministic finite automaton $\mathcal{A}_0 = (Q_0, \Sigma, \delta_0, q_0, Q_0)$ recognising the implementable language L. The set of green transitions can be computed by means of well-known algorithms. The other colours are computed to satisfy the invariants.

Algorithm If the current automaton has no red transitions, then the algorithm stops. Otherwise, it chooses a red transition $q \xrightarrow{a} q'$, and proceeds as follows:

a. Deletes the transition $q \xrightarrow{a} q'$.
b. If there is a state q'' such that $W(q) \cdot a \approx_Z W(q'')$ then the algorithm adds a black transition $q \xrightarrow{a} q''$.
c. Otherwise, the algorithm
  *c*1. creates a new (final) state r,
  *c*2. adds a new green transition $q \xrightarrow{a} r$ (and so $W(r) := W(q) \cdot a$), and
  *c*3. for every transition $q' \xrightarrow{b} q''$, adds a new transition $r \xrightarrow{b} s$ with $s := \delta_0(q_0, W(r) \cdot b)$. The new transition is coloured red if $W(r) \cdot b \not\approx_Z W(s)$ and black otherwise.

Proposition 3. *The algorithm described above always terminates and its output is a safe deterministic finite asynchronous automaton recognising the implementable language L.*

Unfortunately, as we will see later in one of our case studies, the algorithm can produce automata with many more states than necessary. We have implemented a heuristic that allows to 'stop early' if the automaton synthesised so far happens to already be a solution, and otherwise guides the algorithm in the choice of the next red transition.

For this, we need a test that, given a distributed alphabet (Σ, dom) and an automaton $\mathcal{A}$, checks if $\mathcal{A}$ is an asynchronous automaton with respect to dom (i.e., checks the existence of the sets Q_k and the relations δ_a). Moreover, if $\mathcal{A}$ is not asynchronous, the test should produce a "witness" transition of this fact. Fortunately, Morin provides in [Mor98] precisely such a test:

Theorem 5. *[Mor98] Let $\mathcal{A}$ be a deterministic automaton and dom be a distribution. There is the least family of equivalences $(\equiv_k)_{k \in Proc}$ over the states of $\mathcal{A}$ such that (below we denote by $q \equiv_{dom(a)} q'$ if $\forall k \in dom(a) : q \equiv_k q'$)*

DE_1*:* $q \xrightarrow{a} q' \wedge k \notin dom(a) \Rightarrow q \equiv_k q'$

DE_2*:* $q_1 \xrightarrow{a} q_1' \wedge q_2 \xrightarrow{a} q_2' \wedge q_1 \equiv_{dom(a)} q_2 \Rightarrow q_1' \equiv_{dom(a)} q_2'$

Moreover $\mathcal{A}$ is an asynchronous automaton over dom iff the next two conditions hold for any states q, q' and any action a:

DS_1*:* $(\forall k \in Proc : q_1 \equiv_k q_2) \Rightarrow q_1 = q_2$

DS_2*:* $(\exists q_1' : q_1 \xrightarrow{a} q_1' \wedge q_1 \equiv_{dom(a)} q_2) \Rightarrow \exists q_2' : q_2 \xrightarrow{a} q_2'$

It is not difficult to show that the least equivalences $(\equiv_k)_{k \in Proc}$ can be computed in polynomial time by means of a fixpoint algorithm, and so Theorem 5 provides a polynomial test to check if $\mathcal{A}$ is asynchronous.

Because we are interested only in an asynchronous automaton accepting the same language as the initial automaton, a weaker version of Theorem 5 suffices: If the least family of equivalences satisfying DE_1 and DE_2 also satisfies DS_2 (but not necessarily DS_1), there exists an asynchronous automaton recognising the same language as $\mathcal{A}$.

Notice that if $\mathcal{A}$ passes the test then we can easily derive the sets Q_k of local states and the δ_a's functions for every action a: Q_k contains the equivalence classes of $\equiv_k$; given two classes q, q', we have $(q, q') \in \delta_a$ iff $\mathcal{A}$ contains an a-transition between some representatives of q, q'. We remark for later use in our case studies that the proof of [Mor98] proves in fact something stronger than Theorem 5: Any equivalence satisfying DE_1, DE_2, and DS_2 can be used to obtain an asynchronous automaton language-equivalent with $\mathcal{A}$. The least family is easy to compute, but it yields an implementation in which the sets Q_k are too large.

If $\mathcal{A}$ does not pass the test (this implies a red transition involved in the failure), the heuristic will propose a red transition to be processed by the algorithm. We find this transition by applying Morin's test to the subautomaton $\mathcal{A}_{g\&b}$ containing only the green and black transitions of $\mathcal{A}$. There are two cases: (1) the test fails and then we can prove that there is a red edge involved in the failure of DS_2 on $\mathcal{A}_{g\&b}$: $\exists q_1 \xrightarrow{a} q_1'$ green or black and $q_1 \equiv^{g\&b}_{dom(a)} q_2$ and $\exists q_2 \xrightarrow{a} q_2'$ red or (2) the test is successful and then we iteratively add red transitions to the subautomaton $\mathcal{A}_{g\&b}$ until DS_2 is violated. In either case, we find a red transition as a candidate for the unfolding algorithm.

5 Case Study: Mutual Exclusion

A *mutual exclusion* (*mutex* for short) situation appears when two or more processes are trying to access for 'private' use a common resource. A distributed solution to the mutex problem is a collection of programs, one for each process, such that their concurrent execution satisfies three properties: *mutual exclusion* (it is never the case that two processes have simultaneous access to the resource), *absence of starvation* (if a process requests access to the resource, the request is eventually granted), and *deadlock freedom*.

We consider first the problem for two processes. Let the actions be

$$\Sigma := \{req_1, enter_1, exit_1, req_2, enter_2, exit_2\}$$

with the intended meanings: request access to, enter and exit the critical section giving access to the resource. The indices 1 and 2 specify the process that executes the action.

We fix now a distribution. Obviously, we wish to have two processes P_1, P_2 such that $\Delta(P_1) = \{req_1, enter_1, exit_1\}$ and $\Delta(P_2) = \{req_2, enter_2, exit_2\}$. We also want $req_1 \| req_2$ so we need at least two extra processes V_1 and V_2, such that $\Delta(V_1)$ contains req_1 but not req_2, and $\Delta(V_2)$ contains req_2 but not req_1. So let:

$\Delta(V_1) = \{req_1, enter_1, exit_1, enter_2\}$ and $\Delta(V_2) = \{req_2, enter_2, exit_2, enter_1\}$.[4]

[4] We could also add $exit_1$ to $\Delta(V_1)$ and $exit_2$ to $\Delta(V_2)$; the solution does not change.

Next, we define a regular language, $Mutex_1$, specifying the desired behaviours of the system. We want $Mutex_1$ to be the maximal language satisfying the following conditions:

1. $Mutex_1$ is included in the shuffle of prefix-closures of $(req_1\, enter_1\, exit_1)^*$ and $(req_2\, enter_2\, exit_2)^*$.
   I.e., the processes execute $req_i\, enter_i\, exit_i$ in cyclic order.
2. $Mutex_1 \subseteq \Sigma^* \setminus [\Sigma^* enter_1 (\Sigma \setminus exit_1)^* enter_2 \Sigma^*]$ and its dual version.
   I.e., a process must exit before the other one can enter. This guarantees *mutual exclusion*.
3. $Mutex_1 \subseteq \Sigma^* \setminus [\Sigma^* req_1 (\Sigma \setminus enter_1)^* enter_2 (\Sigma \setminus enter_1)^* enter_2 \Sigma^*]$ and dual.
   I.e., after a request by one process the other process can enter the critical section at most once. This guarantees *absence of starvation*.
4. For any $w \in Mutex_1$, there exists an action $a \in \Sigma$ such that $wa \in Mutex_1$.
   This guarantees *deadlock freedom*.

Condition 3 needs to be discussed. In our current framework we cannot deal with 'proper' liveness properties, like: If a process requests access to the critical section, then the access will eventually be granted. This is certainly a shortcoming of our current framework. In this example, we enforce absence of starvation by putting a concrete bound on the number of times a process can enter the critical section after a request by the other process.

The largest language satisfying conditions 1-3 is regular because of the closure properties of regular languages, and a minimal automaton recognising it can be easily computed. Since it is deadlock-free, it recognises the largest language satisfying conditions 1-4. [5]

It turns out that the minimal automaton $\mathcal{A}_1$ for $Mutex_1$ is safe, satisfies ID, FD, and $\Sigma(\mathcal{A}_1) = \Sigma$. Using Proposition 2 the recognised language is implementable. This allows us to apply Zielonka's construction, that yields a safe asynchronous automaton with **34** states. Applying our heuristic based on Morin's test we obtain that the minimal automaton recognising $Mutex_1$ and having **14** states, is already an asynchronous automaton. Families of local states and transitions can be constructed using Morin's theorem. The processes P_1 and P_2 have three local states each, while the processes V_1 and V_2 have 7 states.

We can now ask if the solution can be simplified, i.e., if there is a smaller family of local states making the minimal automaton asynchronous. This amounts to finding a larger family $(\equiv_k)_{(k \in Proc)}$ of equivalences satisfying the properties of Theorem 5. This can be done by merging equivalence classes, and checking if the resulting equivalences still satisfy the properties. We have implemented this procedure and it turns out that there exists another solution in which V_1 and V_2 have only 4 states. Figure 2 (top) shows the resulting asynchronous automaton, translated into pseudocode for legibility. There $\langle com \rangle$ denotes that the command com is executed in one single atomic step. We have represented the processes

[5] If this had not been the case, the largest automaton would have been obtained by removing all states not contained in any infinite path.

Initialization: $v_1 := 0$; $v_2 := 0$

Process 1	**Process 2**
repeat forever	**repeat** forever
[NCS1];	[NCS2];
$v_1 := 1$;	$v_2 := 1$;
$\langle$ **await** ($v_1 \in \{1,3\}$ and $v_2 \in \{0,1\}$) **then** ($v_1 := 2$ and (**if** $v_2 = 1$ **then** $v_2 := 3$)) $\rangle$;	$\langle$ **await** ($v_1 \in \{0,1\}$ and $v_2 \in \{1,3\}$) **then** ((**if** $v_1 = 1$ **then** $v_1 := 3$) and $v_2 := 2$) $\rangle$;
[CS1];	[CS2];
$v_1 := 0$	$v_2 := 0$
end repeat	**end repeat**

Initialization: $v_1 := 0$; $v_2 := 0$

Process 1	**Process 2**
ncs_1: [NCS1];	ncs_2: [NCS2];
$\langle$ **case** ($v_1 = 0$): $v_1 := 1$; **goto** e_1	$\langle$ **case** ($v_2 = 0$): $v_2 := 1$; **goto** e_2
case ($v_1 = 2$): $v_1 := 1$; **goto** e'_1	**case** ($v_2 = 2$): $v_2 := 3$; **goto** e_2
case ($v_1 = 3$): $v_1 := 4$; **goto** e'_1 $\rangle$	e_2: $\langle$ **await** $v_1 \in \{0,2,3,4\}$ **then**
e_1: $\langle$ **await** $v_2 \in \{0,1\}$ **then**	**case** ($v_1 = 0$): $v_1 := 2$; **goto** cs_2
case ($v_2 = 0$): **goto** cs_1	**case** ($v_1 = 2$): $v_1 := 0$; **goto** cs_2
case ($v_2 = 1$): **goto** cs'_1 $\rangle$	**case** ($v_1 = 3$): $v_1 := 2$; **goto** cs_2
e'_1: $\langle$ **await** $v_2 \in \{2,3\}$ **then**	**case** ($v_1 = 4$): $v_1 := 1$; **goto** cs_2 $\rangle$
case ($v_2 = 2$): $v_2 := 0$; **goto** cs_1	cs_2: [CS2];
case ($v_2 = 3$): $v_2 := 1$; **goto** cs'_1 $\rangle$	**case** ($v_2 = 1$): $v_2 := 2$; **goto** ncs_2
cs_1: [CS1]; $v_1 := 0$; **goto** ncs_1	**case** ($v_2 = 3$): $v_2 := 0$; **goto** ncs_2
cs'_1: [CS1]; $v_1 := 3$; **goto** ncs_1	

Fig. 2. The two synthesised solutions for Mutex (N=2)

V_1 and V_2 as two variables with range $[0,1,2,3]$. [6] By construction, the algorithm satisfies mutual exclusion, absence of starvation, and deadlock freedom. Moreover, the two processes can make requests independently of each other.

Using the results of [BD98] it is easy to show that $Mutex_1$ is not a Petri net language. However, it is a product language in the sense of [CMT99]. The results of [CMT99] also allow to derive the solution of Fig. 2. In this case, asynchronous automata do not have an advantage.

5.1 Mutual Exclusion Revisited

The mutex algorithm of the previous section requires to update the variables v_1 and v_2 before entering the critical section in one single atomic action, which is difficult to implement. Is it possible to obtain a solution that avoids this problem? We observe that the problem lies in the distribution we have chosen. We have $\Delta(V_1) \cap \Delta(V_2) = \{enter_1, enter_2\}$, and so both V_1 and V_2 are involved in the *enter* actions, which means that the implementation of both $enter_1$ and $enter_2$ requires to update both of v_1 and v_2 in a single atomic action. So we look for a different distribution in which $\Delta(V_1) \cap \Delta(V_2) = \emptyset$. We take:

$$\Delta(V_1) = \{req_1, enter_2, exit_1\} \text{ and } \Delta(V_2) = \{req_2, enter_1, exit_2\}.$$

[6] The pseudocode was derived by hand, but it would be not difficult to automatise the process.

Unfortunately, $Mutex_1$ is not implementable anymore under this new distribution. The minimal automaton fails to satisfy FD: There is a state in which both $enter_1$ and $enter_2$ are enabled (and $enter_1 \| enter_2$), but there is no converging state to close the diamond. We then apply first heuristic from Sect. 4.1 and we indeed find a subautomaton satisfying ID and FD, deadlock-free and containing all the actions.

Zielonka's construction yields a safe asynchronous automaton with **4799** states. Fortunately, our heuristic yields an asynchronous automaton with only **20** states (see [SEM03]). Once distributed over the four processes of the specification (and merging local states if possible), we obtain the pseudocode shown in Fig. 2 (bottom). The variables v_1 and v_2 range over $[0, 1, 2, 3, 4]$ and $[0, 1, 2, 3]$ respectively. The labels associated with the commands suggest their type, for example r_1 means a request of the first process and x_2 means an exit from the critical section of the second process. Notice that the command corresponding to a label is executed atomically and that the program pointers for the two components advance only as a result of a **goto** command.

The components are now asymmetric, due to the fact that the first heuristic 'solved' the FD conflict by removing an $enter_2$ transition. Yet the algorithm is starvation-free: If the second process request access to the critical section, it will receive it as soon as possible.

The language $Mutex_2$ is neither a Petri net language nor the language of a product of transition systems, and so the procedures of [BD98,CMT99] cannot be applied.

5.2 More Processes

When we consider the mutual exclusion problem for an arbitrary number of processes $N \geq 2$, we choose the alphabet $\Sigma = \cup_{1 \leq i \leq N} \{req_i, enter_i, exit_i\}$. There exist several distributions of the actions. We choose generalizations of the two distribution used for $N = 2$. For $1 \leq i \leq N$:

- $\Delta_1(P_i) := \{req_i, exit_i, enter_1, \ldots, enter_N\}$
- $\Delta_2(P_i) := \{req_i, exit_i, enter_i\}$, $\Delta_2(V_i) = \Delta_1(P_i) \backslash enter_i$

We also generalize the regular specification of the problem. E.g., the mutual exclusion property is specified as: $\Sigma^* \backslash \bigcup_{i \neq j} [\Sigma^* enter_i (\Sigma \backslash exit_i)^* enter_j \Sigma^*]$.

The experiments for $N = 2, 3, 4, 5$ are presented in Table 1. In the first column, we give the parameters of the problem. In the second column, we give the size of the minimal automaton accepting the regular specification together with the number of the processors in the distribution. (The tool AMoRE [Amo] was used to construct the minimal automata recognising the regular specification.) In each of the following columns *size* represents the global state space of the solution (the asynchronous automaton) and *time* is the computation time given in seconds. A dash symbol '–' represents the fact that the system run out of memory without finding a solution. The third and fourth columns give the results after applying the first and respectively second heuristic in Sect. 4.1, followed by Zielonka's procedure. The fifth and sixth columns give the results after applying

Table 1. Experimental results

Problem	Input		Zielonka 1		Zielonka 2		Heuristic 1		Heuristic 2	
	$\|\mathcal{A}\|$	$\|\Delta\|$	size	time	size	time	size	time	size	time
Mutex(2,Δ_1)	14	4	34	<0.01	23	<0.01	14	<0.01	10	<0.01
Mutex(2,Δ_2)	14	4	4799	5.30	2834	2.66	17	<0.01	16	<0.01
Mutex(3,Δ_1)	107	6	–	–	–	–	107	<0.01	30	<0.01
Mutex(3,Δ_2)	107	6	–	–	–	–	–	–	58	0.11
Mutex(4,Δ_1)	1340	8	–	–	–	–	1340	0.31	62	1.25
Mutex(4,Δ_2)	1340	8	–	–	–	–	–	–	157	3.83
Mutex(5,Δ_1)	25338	10	–	–	–	–	25338	170.95	147	1000.76
Mutex(5,Δ_2)	25338	10	–	–	–	–	–	–	387	1053.79

the first and respectively second heuristic in Sect. 4.1, followed by the heuristic in Sect. 4.2. (The experiments were performed on a machine with 2.4 GHz CPU and 1 GB RAM.)

6 Further Remarks

We have proposed to apply the theory of asynchronous automata to the problem of synthesising closed distributed algorithms. We have observed that the right implementation model are safe asynchronous automata, and we have characterised their languages. We defined the synthesis problem in our framework and proved that it is undecidable, therefore we focused our attention on an NP-complete subclass of solutions. We have implemented Zielonka's algorithm, and observed that it leads to large implementations even for natural and relevant case studies where much smaller implementations exist. We have derived heuristics to make the synthesis problem more feasible in practice. We have used the heuristics to automatically generate mutual exclusion algorithms.

Obtaining 'Elegant' Solutions: Our solutions to the mutex problem are not 'elegant': They use variables with larger domains than those appearing in the literature, and a human finds it difficult to understand why they are correct. Notice, however, that this is the case with virtually all computer generated outputs, whether they are HTML text, program code, or a computer generated proof of a formula in a logic. Our solutions are correct and relatively small.

Specifying with Temporal Logic: Notice that our approach is compatible with giving specifications as LTL temporal logic formulas over finite strings, since the language of finite words satisfying a formula is known to be regular, and an automaton recognising this language can be effectively computed.

Dealing with Liveness Properties: Currently our approach cannot deal with liveness properties. Loosely speaking, 'eventually' properties have to be transformed into properties of the form 'before this or that happens'. Dealing with liveness properties requires to consider the theory of asynchronous automata on infinite words, for which not much is known yet (see Chap. 11 of [DR95]). The approaches of [BD98,CMT99] take a transition system as specification, and so

do not consider liveness properties either. The approach of [MT02] can deal with liveness properties, but it can only synthesise controllers satisfying certain conditions (clocked controllers). These conditions would not appear in a reformulation of our results in a distributed controllers synthesis.

Acknowledgments. We thank Volker Diekert, Rémi Morin, Madhavan Mukund, Holger Petersen, and several anonymous referees for useful comments. This work was partially supported by EPSRC grant GR64322/01.

References

[Amo] AMoRE. http://www-i7.informatik.rwth-aachen.de/d/research/amore.html

[BD98] E. Badouel and Ph. Darondeau. Theory of regions. In *Lectures on Petri Nets I (Basic Models)*, LNCS 1491 (1998) 529–588.

[CMT99] I. Castellani, M. Mukund, and P.S. Thiagarajan. Synthesizing distributed transition systems from global specifications. In *Proc. FSTTCS* 19, LNCS 1739 (1999) 219–231.

[DR95] V. Diekert and G. Rozenberg (Eds.) *The Book of Traces*. World Scientific, 1995.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming* 2 (1982) 241–266.

[KV01] O. Kupferman and M.Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, (2001).

[Maz87] A. Mazurkiewicz. Trace theory. In LNCS 255 (1987) 279–324.

[Mor98] R. Morin. Decompositions of asynchronous systems. In *CONCUR'98*, LNCS 1466 (1998) 549–564.

[MT02] P. Madhusudan and P.S. Thiagarajan. A decidable class of asynchronous distributed controllers. In *Proc. CONCUR'02*, LNCS 2421 (2002) 145–160.

[Muk02] M. Mukund. From global specifications to distributed implementations. In *Synthesis and Control of Discrete Event Systems*, B. Caillaud, P. Darondeau, L. Lavagno (Eds.), Kluwer (2002) 19–34.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic. *ACM TOPLAS* 6(1) (1984) 68–93.

[PR89] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th ICALP*, LNCS 372 (1989) 652–671.

[SEM03] A. Ştefănescu, J. Esparza, and A. Muscholl. Synthesis of distributed algorithms using asynchronous automata. Available at: http://www.fmi.uni-stuttgart.de/szs/publications/stefanan/concur03_full.ps

[Zie87] W. Zielonka. Notes on finite asynchronous automata. *R.A.I.R.O. Inform. Théor. Appl.* 21 (1987) 99–135.

[Zie89] W. Zielonka. Safe executions of recognizable trace languages by asynchronous automata. In LNCS 363 (1989) 278–289.