Decomposing Port Automata

Christian Koehler*

CWI, P.O. Box 94079
1090 GB Amsterdam
The Netherlands

christian.koehler@cwi.nl

Dave Clarke

Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium

dave.clarke@cs.kuleuven.be

ABSTRACT

Port automata are an operational model for component connectors in a coordination language such as Reo. They describe which sets of ports can synchronize in each state of the connector being modelled. This paper presents decomposition theorems for port automata, namely that all (finite) port automata can be generated from a small set of primitive port automata. Applying these results to component connectors means that all component connectors can be constructed from just *two* primitive connectors.

1. INTRODUCTION

The coordination paradigm [6] divides software systems into two orthogonal aspects: the *computation* which is performed by a number of independent components, and the *coordination* of these components using some kind of connector. Coordination models describe the interaction or, in distributed environments, the communication between the components. A recent approach to coordination is exemplified in the coordination model Reo [1], wherein component connectors are constructed by composing primitives such as channels to form circuit-like connectors. Port automata are an operational model for such component connectors. This automata model describes the synchronisation on sets of *ports*, depending on the internal state of the connector. The model abstracts from both the direction and contents of data flow. Port automata models of larger connectors are constructed using a *product*-operation, which synchronizes connected ports, and a *hiding*-operation, which removes (direct) observations on internal ports (as in CCS [9]).

This work investigates decomposition properties of these automata. In particular, any (finite) port automaton can be decomposed into the composition (using product and hiding) of automata from a small finite set consisting of an exclusive router (XOR) and one of two variations of an asynchronous buffer (a Flip-Flop or FIFO).

*Supported by NWO/GLANCE projects WoMaLaPaDiA and SYANCO.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...\$5.00.

We make the following contributions in this paper. We show that any finite port automaton can be generated from a finite collection of primitive automata. This decomposition is achieved by examining port automata directly—there is no need to consider the language generated by the automaton. In the context of component connectors, our result means that a finite number of channel types is sufficient to implement every behaviour expressible in port automata.

Paper structure. The rest of the paper is organized as follows. Section 2 introduces port automata. Section 3 contains our decomposition results. Section 4 contains related work and finally Section 5 contains concluding remarks and possible future research directions.

2. PORT AUTOMATA

This section defines port automata, their product and hiding operations, and an equivalence relation on them. As examples, we model some basic Reo channels using port automata.

Definition 1. A port automaton $\mathcal{A} = (\mathcal{Q}, \mathcal{N}, \rightarrow, \mathcal{Q}_0)$ consists of a set of states $\mathcal{Q}$, a set of ports $\mathcal{N}$, a transition relation $\rightarrow \subseteq \mathcal{Q} \times 2^{\mathcal{N}} \times \mathcal{Q}$ and a set of initial states $\mathcal{Q}_0 \subseteq \mathcal{Q}$.

We write transitions as $q \xrightarrow{N} p$ with $q, p \in \mathcal{Q}$ and $N \subseteq \mathcal{N}$. The interpretation is that there is data flow at the ports N and no data flow at the rest of the ports $\mathcal{N} \setminus N$. The model permits τ -transitions, namely whenever $N = \emptyset$. Hence, there can be silent steps without any data flow. For simplicity, we omit the curly brackets around the ports in figures.

Consider the port automata in Figure 1, which correspond to basic Reo channels. The *Sync*(a, b) automaton models a primitive that allows only synchronous activity on ports a and b . The *Lossy*(a, b) additionally can have activity just on

Figure 1: Examples of port automata.

port a . The $XOR(a, b, c)$ synchronizes a with either b or c . Finally, the $FIFO(a, b)$ automaton models a unary buffer.

Now we introduce the two main operations on port automata. These are product and hiding.

Definition 2. The product of two port automata $\mathcal{A}_1 = (\mathcal{Q}_1, \mathcal{N}_1, \rightarrow_1, \mathcal{Q}_{0,1})$ and $\mathcal{A}_2 = (\mathcal{Q}_2, \mathcal{N}_2, \rightarrow_2, \mathcal{Q}_{0,2})$ is defined by

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, \mathcal{N}_1 \cup \mathcal{N}_2, \rightarrow, \mathcal{Q}_{0,1} \times \mathcal{Q}_{0,2})$$

where $\rightarrow$ is defined by the rule

$$\frac{q_1 \xrightarrow{N_1}_1 p_1 \quad q_2 \xrightarrow{N_2}_2 p_2 \quad N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2} \langle p_1, p_2 \rangle}$$

and the following and its symmetric rule

$$\frac{q_1 \xrightarrow{N_1}_1 p_1 \quad N_1 \cap \mathcal{N}_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N_1} \langle p_1, q_2 \rangle}$$

The first rule models the synchronous product of two transitions. The second and the third rule are for an interleaving of actions. Note that, for applying any of the rules, every port shared by the two automata must be either enabled or disabled in both transitions. The following definition introduces hiding of ports.

Definition 3. Let $\mathcal{A} = (\mathcal{Q}, \mathcal{N}, \rightarrow, \mathcal{Q}_0)$ be a port automaton and $M \subseteq \mathcal{N}$. The port automaton $\mathcal{A} \setminus M$ is defined by $\mathcal{A} \setminus M = (\mathcal{Q}, \mathcal{N} \setminus M, \dashrightarrow, \mathcal{Q}_0)$ where $\dashrightarrow$ is given by:

$$q \dashrightarrow p \Leftrightarrow q \xrightarrow{N} p$$

The hiding operation simply removes all occurrences of a set of ports. The following example shows how these two operations can be used to build more complex port automata.

When using predefined primitives, we will instead just write:

$$(FIFO(a, b) \bowtie XOR(a, c, d)) \setminus \{a\}$$

To reason about the constructions we further need a notion of behavioral equivalence. For this purpose, we define a notion of bisimulation between two port automata.

Definition 4. Given port automata $\mathcal{A}_1 = (\mathcal{Q}_1, \mathcal{N}, \rightarrow_1, \mathcal{Q}_{0,1})$ and $\mathcal{A}_2 = (\mathcal{Q}_2, \mathcal{N}, \rightarrow_2, \mathcal{Q}_{0,2})$ a bisimulation is a relation $\sim \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ with the following properties:

1. Let $q_1 \in \mathcal{Q}_{0,1}$ and $q_1 \xrightarrow{N}_1 p_1$. Then there exists a $q_2 \in \mathcal{Q}_{0,2}$ with $q_2 \xrightarrow{N}_2 p_2$ and $p_1 \sim p_2$, and vice versa.
2. If $q_1 \sim q_2$ then for all transitions $q_1 \xrightarrow{N}_1 p_1$ there exists $q_2 \xrightarrow{N}_2 p_2$ such that $p_1 \sim p_2$, and vice versa.

Note that this notion of bisimulation is *late*, in the sense that the initial states are not part of the relation. If there is a bisimulation that relates two port automata we denote this by $\mathcal{A}_1 \sim \mathcal{A}_2$. Every bisimulation is a congruence, i.e., it is compatible with the product and hiding operation:

THEOREM 1. *Compatibility with join and hiding:*

1. $\mathcal{A}_1 \sim \mathcal{A}_2$ and $\mathcal{A}'_1 \sim \mathcal{A}'_2 \Rightarrow (\mathcal{A}_1 \bowtie \mathcal{A}'_1) \sim (\mathcal{A}_2 \bowtie \mathcal{A}'_2)$.
2. $\mathcal{A}_1 \sim \mathcal{A}_2 \Rightarrow (\mathcal{A}_1 \setminus B) \sim (\mathcal{A}_2 \setminus B)$.

PROOF. For showing 1. we observe that the relation

$$\langle q_1, q'_1 \rangle \sim \langle q_2, q'_2 \rangle \Leftrightarrow q_1 \sim q_2 \text{ and } q'_1 \sim q'_2$$

defines a bisimulation on the product automata. For 2., any bisimulation is still valid after a hiding of ports. $\square$

In the following section we show how port automata can be decomposed. We use the term *decomposition* in the sense that a port automaton can be written as a finite product of other automata with the optional hiding of some ports.

3. DECOMPOSITION

In this section we show how port automata can be decomposed. First, we show that *stateless* port automata, i.e., port automata with just one state, can be decomposed into a product of XORs. Second, we extend the result to arbitrary port automata by including either the FIFO or a so-called Flip-Flop primitive.

Being stateless port automata, the *Lossy* and *Sync* in Figure 1 can be decomposed into XORs using the following constructions:

$$Lossy(a, b) = XOR(a, b, c) \setminus \{c\}$$

$$Sync(a, b) = (XOR(a, x, y) \bowtie XOR(b, x, y)) \setminus \{x, y\}$$

Both automata consist only of a single state and hence, model stateless connectors (cf. [4, 5]). The first step in the decomposition process is a construction which shows that every stateless connector can be constructed from XORs. We consider two ways of composing XORs, which can be depicted in a *connector notation* as in Figure 2.

Figure 2: Composing XORs.

In this notation, (binary) XORs are represented as $\neg\angle$, while hidden and normal ports are depicted by $\circ$ and $\bullet$, respectively. With the first kind of composition, depicted in 2a), one can build n -ary XORs out of binary ones. Adding

XORs in this way *splits* a transition $q \xrightarrow{N \cup \{x\}} q$ into two transitions $q \xrightarrow{N \cup \{b_2\}} q$ and $q \xrightarrow{N \cup \{b_3\}} q$. This essentially models the *mutual exclusion* of ports b_2 and b_3 . In the following we will use the notation $XOR(a, B)$ for an n -ary XOR with $B = \{b_1, \dots, b_n\}$ a set of ports.

The second way of composing XORs is shown in 2b). This construction uses the internal ports to synchronize the external ports. In this particular example, a and b can be activated only together, either via x or y . The port automaton coincides with the *Sync*(a, b), as mentioned above.

We use construction *a*) to generate n -ary XORs and *b*) to synchronize ports. In combination, these can be used to construct any stateless automaton. Given a stateless port automaton $\mathcal{A} = (\{q\}, \mathcal{N}, \rightarrow, \{q\})$ with $\mathcal{N} = \{a_1, \dots, a_n\}$ and with transtions $q \xrightarrow{N_i} q$ for $1 \leq i \leq m$. The following construction yields a composition of XORs equivalent to $\mathcal{A}$:

1. Introduce a new port x_i for each transition $q \xrightarrow{N_i} q$. Let $X = \{x_1, \dots, x_m\}$.
2. Introduce a new port δ and create $XOR(\delta, X)$.
3. For every port $a \in \mathcal{N}$ consider the following set
$$X_a = \{x_i \in X \mid \exists (q \xrightarrow{N_i} q) : a \in N_i\}$$
and create $XOR(a, X_a)$.
4. Join all XORs and hide δ and all elements of X .

Consider the following example decomposition of a stateless port automaton.

The connector consists of four XORs: one for each port a_1, a_2, a_3 and one for the port δ . Furthermore, there is a one-to-one correspondence between the internal ports $X = \{x_1, x_2, x_3\}$ and the transitions in $\mathcal{A}$, i.e., x_1 represents the transition $\{a_1\}$, x_2 represents $\{a_1, a_2\}$ and finally x_3 stands for the transition $\{a_1, a_2, a_3\}$. The correctness of this construction can be verified using the following argument:

- The ports of the constructed automaton are the same as in $\mathcal{A}$, as δ and all ports in X have been hidden. The automaton is also stateless.
- Every port $x_i \in X$ stands for a transition $q \xrightarrow{N_i} q$ and
$$a \in N_i \Leftrightarrow x_i \in X_a,$$
i.e., port x_i is enabled exactly when all $a \in N_i$ are enabled. That is, x_i synchronizes the ports of N_i .
- An $x_i \in X$ can be enabled only if δ is enabled and when δ is enabled exactly one $x_i \in X$ is enabled. Hence, no concurrent activation of the port sets N_i is possible.

Using this construction we have shown how to decompose stateless port automata into XORs. We consider now the full class of port automata and start again with an example.

Figure 3: Sequencer.

The Sequencer (cf. [1]), shown in Figure 3, can be constructed using just FIFOs. If we would try to build a connector with this behaviour, we could simply compose n FIFO

channels in a loop and make the last one *full*, i.e., it already carries a token at the beginning. Intuitively, the token moves from one FIFO channel to another in one step and starts from the beginning after n steps. In the port automata model, the difference between empty and full FIFOs is realized by switching the ports, so that $FIFO(b, a)$ models a full FIFO from a to b . Hence, we can construct the n -ary Sequencer in the port automata model as follows:

$$\begin{aligned} Seq(a_1 \dots a_n) &= FIFO(a_1, a_2) \bowtie FIFO(a_2, a_3) \bowtie \\ &\dots \bowtie FIFO(a_{n-1}, a_n) \bowtie FIFO(a_1, a_n) \end{aligned}$$

where the last FIFO is full. Note that this product also introduces some unreachable states not shown in Figure 3.

The general decomposition of an arbitrary port automaton involves the following steps, which do not change the behavior of the automaton—up to bisimulation (Definition 4). The first step reduces the number of initial states to one. The second step removes all parallel transitions, and labels every transition uniquely. The third step breaks the resulting automaton into the product of port automata each containing not more than two states and four transitions. There is only a finite number of such automata; we show that the largest of these, which we will call *Flip-Flop*, suffices. The last step reduces this automaton to XORs and FIFOs, completing the decomposition.

1) Reducing initial states. The first step replaces the automaton by one that has exactly one initial state. Firstly, introduce a new initial state q' . Then for each initial state q_0 and each transition $q_0 \xrightarrow{N} p$, introduce a new transition $q' \xrightarrow{N} p$. Finally, set the initial state set to be $\{q'\}$. Figure 4 gives an example of this construction, where the bisimulation according to Definition 4 is $\{q_0 \sim p_0, q_1 \sim p_1\}$.

Figure 4: Reducing number of initial states.

2) Removing parallel transitions. Given an arbitrary port automaton $\mathcal{A}$, construct a new port automaton $\mathcal{A}'$ which has the same number of (initial) states. For all parallel transitions between each pair of states

in $\mathcal{A}$ introduce a single transition $q_0 \xrightarrow{\{x\}} q_1$ in $\mathcal{A}'$ where x is a fresh port. The port set of $\mathcal{A}'$ consists exactly of these ports: $\mathcal{N}' = \{x_1, \dots, x_m\}$. For x corresponding to transitions with labels $N_1, \dots, N_n$, construct the following

stateless port automaton:

Joining these stateless port automata with $\mathcal{A}'$ and hiding the newly introduced ports yields the original automaton

$$\mathcal{A} \approx (\mathcal{A}' \bowtie \mathcal{A}_{x_1} \bowtie \dots \bowtie \mathcal{A}_{x_m}) \setminus \mathcal{N}'$$

up to a change of state names. The stateless automata $\mathcal{A}_{x_i}$ can be further decomposed into XORs, as described above. The automaton $\mathcal{A}'$ has the property that every port occurs alone in exactly one transition. We will call an automaton with this property *singleton* port automaton.

3) Splitting singleton port automata. Given a singleton port automaton $\mathcal{A} = (\mathcal{Q}, \mathcal{N}, \rightarrow, \mathcal{Q}_0)$ with more than 2 states, the following steps decompose it into two automata, one with 2 states, the other with $\lceil |\mathcal{Q}|/2 \rceil$ states.

If the number of states in the automaton is odd, introduce an unreachable dummy state to make the total number of states even. Let n be the new total number of states. Now organize the states into a $2 \times (n/2)$ grid with dimensions $\{a, b\}$ and $\{1, 2, \dots, n/2\}$, where position $(a, 1)$ is reserved for the initial state, and the remainder of the states of $\mathcal{A}$ are placed arbitrarily in the grid. Now construct two automata

$$\begin{aligned} \mathcal{A}_1 &= (\{a, b\}, \mathcal{N}, \rightarrow_1, \{a\}), \\ \mathcal{A}_2 &= (\{1, \dots, n/2\}, \mathcal{N}, \rightarrow_2, \{1\}) \end{aligned}$$

using the following rule:

$$\frac{\langle i, j \rangle \xrightarrow{\{x\}} \langle i', j' \rangle \quad i, i' \in \{a, b\} \quad j, j' \in \{1, \dots, n\}}{i \xrightarrow{\{x\}}_1 i' \quad j \xrightarrow{\{x\}}_2 j'}$$

It is clear from the construction that the product of the two automata produced is bisimilar to the original automaton and that they are both again singleton automata.

By repeating this step one can decompose a singleton port automaton into a product of singleton port automata with not more than two states. By applying step 2) to these automata the number of transitions can be reduced to at most four, since all parallel transitions are removed. Moreover, these automata have exactly one initial state, since we reduced the number of initial states already in step 1). The *largest* of these two-state automata, i.e., the one with the most transitions is depicted in Figure 5.

Figure 5: A Flip-Flop.

All other possible automata of this shape can be constructed by removing transitions from this one. Transitions can be removed by first creating a new transition using an

XOR and then synchronizing it with the transition to be removed. An example of this construction follows:

$$\begin{aligned} \text{Diagram of Flip-Flop} &= (\text{Flip-Flop}(a, b, x, d) \bowtie \\ &\quad \text{XOR}(x, c, c') \bowtie \text{Sync}(c', d)) \setminus \{x\} \end{aligned}$$

Hence, an arbitrary port automaton can be decomposed into Flip-Flops and XORs. Note also that the number of required Flip-Flops is $\lceil \log n \rceil$, where n is the number of states of the original automaton to be decomposed. However, as the Flip-Flop does not correspond to a channel in Reo, we decompose it now further into ordinary FIFOs.

4) Removing self loops. The final decomposition step replaces all occurrences of Flip-Flops by ordinary FIFOs. The main difference between a Flip-Flop and a FIFO is that FIFOs have no self loop transitions. Figure 6 shows how to *unroll* self loops.

Figure 6: Unrolling self loops.

A more general version of this automaton, where each transition is uniquely labelled, and its further decomposition are depicted in Figure 7. The original can be regained by composing it with the following XORs: $\text{XOR}(a, a_1, a_2)$, $\text{XOR}(b, b_1, b_2)$, $\text{XOR}(c, c_1, c_2)$ and $\text{XOR}(d, d_1, d_2)$, and hiding $\{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2\}$.

Figure 7: Decomposing the unrolled automaton.

Each of the four remaining primitives in Figure 7 can be constructed using a FIFO and two binary XORs. For example, the top left hand automaton is equivalent to

$$(\text{FIFO}(c, d) \bowtie \text{XOR}(c, a_1, c_1) \bowtie \text{XOR}(d, a_2, d_1)) \setminus \{c, d\}.$$

Note that this step is an instance of a more general construction, wherein any port automaton without self loops can be directly constructed from FIFOs and XORs which

more or less encode the automaton structure using the FIFO buffers.¹ However, the number of FIFOs required in this construction is linear in the number of states and hence the resulting connector becomes unnecessarily large in practise.

Summing up, we have shown that arbitrary port automata can be decomposed into XORs and FIFOs. The number of required FIFOs is $4\lceil\log n\rceil$, since we need four FIFOs to encode one Flip-Flop. For example, an n -Sequencer (Figure 3) requires n FIFOs in the original construction, or $\lceil\log n\rceil$ Flip-Flops or $4\lceil\log n\rceil$ FIFOs using our techniques. For an 8-Sequencer, the required stateful primitives are 8 (original), 3 (using Flip-Flops), and 12 (after encoding Flip-Flops), whereas for a 16-Sequencer, the numbers are 16, 4, and 16, after which even our encoding using FIFOs becomes superior to the original.

4. RELATED WORK

Decomposition results for (deterministic) finite automata have a very long history, starting with the famous Krohn-Rhodes Theorem [7]. The primitive automata underlying the Krohn-Rhodes Theorem include a Flip-Flop, similar to but not the same as ours. By treating the labels of our automata as letters of the alphabet, these results apply to our automata, but do not give the results presented here. The key difference is that our labels have structure and our product and hiding operations manipulate that structure.

Port automata abstract away the data constraints from the constraint automata model of Reo [3]. Arbab et al. [2] show how to synthesise Reo connectors from constraint automata, based on a small set of Reo primitives (hence, a small set of constraint automata). The approach, however, takes the bizarre side step through scheduled-data expressions (effectively regular expressions for constraint automata), and hence the resulting connectors are large and unwieldy, and the resulting automata are *not* bisimilar to the original automata. As their approach deals with data, a larger number of primitives (9) are required. Our approach directly encodes automata in terms of other automata, which could then be represented as Reo connectors, the number of primitives is smaller (2), and the encoding produces a bisimilar result.

An algebra for stateless connectors [4] was suggested by Bruni et al. The authors give a completeness result by showing that a finite number of primitives is sufficient to model all stateless connectors of Reo in this framework, although their axiomatization is not finite. The two main differences between the two models is that we do not consider the direction of data flow and that the approach of Bruni is restricted to stateless connectors only.

Expressions in Milner's SCCS [8] have actions corresponding to a multiset of atomic actions (elements of an Abelian group). Synchronisation is built into the model automatically via the group's product operation. Hiding of names corresponds to matching actions with their co-action (inverse in the group). Synchronisation in our model is different, based on the correspondence of names in the two automata being joined. We are unaware of any decomposition theorems for SCCS.

5. CONCLUSION

The use of the decomposition process is twofold. At first glance, it is an expressiveness result, namely that two primitives are sufficient to construct every possible port automaton. The second result is the compactness of the decomposed automaton: the number of stateful primitives depends only logarithmically on the number of states. Due to this, the approach is well suited for synthesising connectors from behavioural specifications. As future work, we plan to extend the results to (causal) constraint automata, thus including both the direction of data flow (and causality issues arising) and the contents of data via data constraints.

6. REFERENCES

- [1] F. Arbab. Reo: a Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14:329–366, 2004.
- [2] F. Arbab, C. Baier, F. S. de Boer, J. J. M. M. Rutten, and M. Sirjani. Synthesis of Reo Circuits for Implementation of Component-Connector Automata Specifications. In *COORDINATION 2005*, volume 3454 of *Lecture Notes in Computer Science*, pages 236–251. Springer, 2005.
- [3] F. Arbab, C. Baier, J. Rutten, and M. Sirjani. Modeling Component Connectors in Reo by Constraint Automata. *Science of Computer Programming*, 61:75–113, 2006.
- [4] R. Bruni, I. Lanese, and U. Montanari. A basic Algebra of Stateless Connectors. *Theoretical Computer Science*, 366(1):98–120, 2006.
- [5] D. Clarke, D. Costa, and F. Arbab. Connector Colouring I: Synchronisation and Context Dependency. *Science of Computer Programming*, 66(3):205–22, 2007.
- [6] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Commun. ACM*, 35(2):97–107, 1992.
- [7] K. Krohn and J. Rhodes. Algebraic Theory of Machines I: Prime Decomposition Theorems for Finite Semigroups and Machines. *Transactions of the American Mathematical Society*, 116:450–464, 1965.
- [8] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
- [9] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

¹Based on an idea of Christel Baier.