Checking Equivalence for Reo Networks

Tobias Blechmann[a,1] Christel Baier[a,1]

[a] *Chair for Algebraic and Logical Foundations of Computer Science*
Technical University of Dresden
Dresden, Germany

Abstract

Constraint automata have been used as an operational model for component connectors described in the coordination language Reo which specifies the cooperation and communication of the components by means of a network of channels. This paper addresses the problem of checking equivalence of two Reo networks. We present a compositional approach for the generation of a symbolic representation of constraint automata for Reo networks and report on an implementation that realizes a partitioning splitter technique for checking bisimulation equivalence for Reo networks. Using a special operator on our symbolic data structure enables efficient treatment of the rich labeled transitions in constraint automata. In order to show the power of this approach we then present some benchmarks.

Keywords: Reo, bisimulation, coordination, partitioning splitter, symbolic model checking

1 Introduction

Reo is a channel-based exogenous coordination language [1]. It was invented to provide the glue code for describing how component instances communicate with each other and how they are coordinated. Since then it has been used to model many different complex systems. The Reo point of view is that a system consists of component instances which execute at different locations and communicate through connectors. The main idea is to define a small set of simple channels and their behavior. More complex connectors then can be constructed through composition of these simple channels. During design of such coordination protocols questions like whether two specifications have the same observable behavior or whether one is an abstraction of another one arise naturally and frequently. Therefore we use *constraint automata* which are a special kind of labeled transition systems introduced by [2]. Constraint automate provide an operational semantics for Reo which nicely reflects its compositional channel-based approach. The model of constraint automata is equipped with operators which mimic the operations provided for Reo. This yields a compositional approach to construct the corresponding constraint automata to a given Reo circuit is provided.

[1] The authors are supported by the EU-project Credo and the DFG-and-NWO-project SYANCO

Notions of bisimulation equivalence for constraint automata yield a definition for the equivalence of constraint automata. This is e.g. important to replace a given Reo connector component by a functional equivalent, but simpler (or cheaper) one. Furthermore bisimulation equivalence can be used as specification formalism: Having a certain coordination mechanism in mind, it is often quite easy to depict an automaton that describes its permissible behavior. In this sense this automaton serves as a specification for the Reo circuit we want to design. Correctness can then be understood w.r.t. bisimulation equivalence. Algorithms to compute bisimulation equivalence classes provide a sound way to check for equivalence of two given constraint automata or Reo circuits.

In this paper we report an implementation of a bisimulation checker using a symbolic approach working on switching functions. One could also try to model the behavior of Reo circuits using well-known bisimulation model checking tools like [3] or [4]. But representing the compositional approach of Reo using standard tools would require to provide a semantics for Reo based on the process algebra operators and therefore leads to huge and rather artificial looking system descriptions. For us the two main challenges were (1) to find an efficient way to deal with the rich labeled transitions occurring in constraint automata and (2) to symbolically compute logical equivalence classes as this is needed for implementation of the partitioning splitter algorithm. We report on how to accomplish the composition of two symbolic representations of constraint automata in a very efficient way.

Then we introduce a pattern equivalence operator which allows to efficiently compute logical equivalence classes and show how it enables a symbolic implementation of the partitioning splitter algorithm. This leads to a tool which enables us to automatically and efficiently treat constraint automata with lots of states but rather few bisimulation equivalence classes. In order to measure the efficiency of our approach we report on some benchmark results.

This paper starts with a summary of the main concepts of Reo and constraint automata in section 2. Section 3 explains how the compositional approach to generate a constraint automaton from a given Reo circuit can work symbolically. Then section 4 explains the main features of our implementation and section 5 reports on our experimental results.

2 Reo and Constraint Automata

We briefly summarize the main concepts of Reo and constraint automata. For further information see [1] and [2]. In Reo every connector is constructed out of channels using the provided operators for channel composition. The simplest channel is a synchronous channel shortly called sync channel. It accepts data written to its source end when the same data can leave the channel at the same moment through its sink end. There are two basic operations used for composition called *join* an *hiding*. *Join* plugs to channel-ends together creating a node at the point of connection. To this node one can connect more channels via join afterwards. If more than one accepting channel-ends are connected to a node every incoming message is simultaneously written to all outgoing channels whenever all outgoing channels at the node are ready to accept data. Whenever more than one channel-end offers

data at a node a non-deterministic choice decides which data is taken and written to all outgoing channels. The *hiding* operation hides away one node which means that the data-flow occurring at this node cannot be observed from outside and no new channel-end can be connected to this node.
Constraint automata as introduced by [2] can serve as an operational semantics for Reo. Similar to other finite automata, constraint automata consist of states and a transition relation between those states. All transitions in constraint automata are labelled to describe the node where data enters a channel or is taken from a channel. The labeling also determines what data item is transferred. Constraint automata use a finite set $\mathcal{N}$ of *names*. Roughly speaking the elements in $\mathcal{N}$ are the nodes in the corresponding Reo circuit. Transitions in constraint automata are labeled with pairs consisting of a non-empty subset $N \subseteq \mathcal{N}$ and a data constraint g. *Data constraints* are propositional formulas describing the data items that enable a transition in the constraint automata. These formulas are built out of atoms d_A where $d \in Data$ a finite data domain and $A \in \mathcal{N}$. To simplify the notations we assume that $Data = \{0, 1\}$. The atom d_A stands for the interpretation 'd is written to or read from port/node A.'. Data constraints are given by the grammar $g ::= true \mid d_A \mid g_1 \vee g_2 \mid \neg g$ where $A \in \mathcal{N}$ is a name and $d \in Data$. The boolean operators $\wedge$ (conjunction), $\oplus$ (exclusive or), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence) can be derived as usual. We often use derived data constraints such as $d_A = d_B$ which means that on port A and port B the same element out of *Data* is transferred. In the sequel we write $DC(N, Data)$ for $N \subseteq \mathcal{N}$ and $N \neq \emptyset$ to denote the set of data constraints that use only atoms $d_A = d$ for $A \in N$. For all data constraints in the whole automaton we write DC as an abbreviation for $DC(\mathcal{N}, Data)$.

Definition 2.1 *(Constraint Automata)* A constraint automaton is a tuple $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, Q_0)$ where Q is the set of states, $\mathcal{N}$ is the set of nodes, $\rightarrow$ is a subset of $Q \times 2^{\mathcal{N}} \times DC \times Q$ called the transition relation of $\mathcal{A}$ and $Q_0 \subseteq Q$ is the set of initial states. We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \rightarrow$. N is called the name-set and g the guard of the transition. We require that $N \neq \emptyset$ and $g \in DC(N, Data)$. $\mathcal{A}$ is called finite iff Q and $\rightarrow$ are finite.[2]

For an intuitive interpretation one can see the states as representations of the connector configurations and the transitions as the possible single-step behavior.

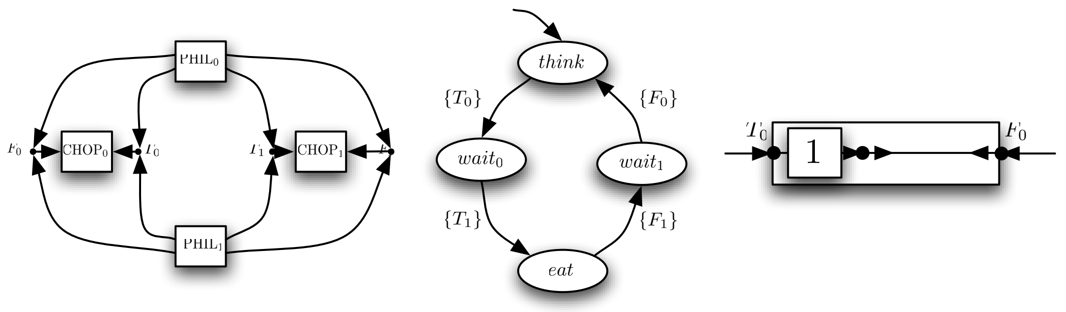


Fig. 1. Two Reo connectors and a constraint automata

[2] We demand *Data* and Q to be finite so here $\rightarrow$ is finite.

In Figure 1 we present the dining philosophers example as shown in [5]. The left part of Figure 1 shows a Reo circuit which specifies the dining philosophers protocol for two philosophers and two chopsticks. Solid arrows stand for sync channels. To gain a constraint automaton that describes the behavior of this circuit we have to specify what happens inside of a PHIL and a CHOP component. The constraint automaton in the middle of the picture models the behavior of one PHIL component. The philosophers starts in state *think* and if this is possible writes [3] a message to T_0 which intuitively means picking one of the chopsticks next to him. After writing the message he enters state $wait_0$. If writing the message T_1 is possible this message is written and the component enters state *eat*. On the way back to state *think* the component then has to write two messages which release the chopsticks. What remains is to specify the behavior of the chopsticks. The right part of Figure 1 shows a Reo circuit for the chopsticks. We use a blocking FIFO1 channel (marked by a box in the middle of the channel) and join its sink end with one of the source ends of a synchronous drain (SYNCDRAIN) channel. The blocking FIFO1 channel behaves in a natural way. If its buffer is empty it accepts one data item. Afterwards it does not accept any write operations to its source end and waits for a read operation happening at the sink end. The SYNCDRAIN channel has two source ends and no sink end. It accepts any data item on one of its ends iff at the same moment another data item is written to the other end. As we are not interested in the communication taking place at the inner node we can hide it which is shown by the enclosing box. The semantics of the two atomic channels we used must again be specified using constraint automata. What remains is to show how we can mimic the join and the hiding operation on the automata level.

Due to the fact that in constraint automata there is no way to distinguish between incoming and outgoing ports we have to use two different operations to mimic the merge semantics of Reo nodes. On the automata level we first provide the product automata operator for joining a source node with another node (no matter what type). Then we use a merger automaton to simulate the merger behavior of multiple joined sink channel ends. This merger automaton can be regarded as a new primitive connector in the Reo circuit plugged before the old node which just performs a non-deterministic choice on its source ports and forwards the chosen data to the old node. Its semantics can be represented by a constraint automaton with only one state and self-loop transitions for every incoming channel-end.

Assume two Reo circuits with node-sets $\mathcal{N}_1$ and $\mathcal{N}_2$ and their corresponding automata are given. We want to perform a join operation for node-pairs $\langle B_i, \bar{B}_i \rangle \in \mathcal{N}_1 \times \mathcal{N}_2,\ i = 1, \ldots, k$ where, for any i, at least one of the nodes B_i or $\bar{B}_i$ is a source node. We assume that the nodes are renamed in a way that $B_1 = \bar{B}_1, \ldots, B_k = \bar{B}_k$ holds and the two automata do not contain other common nodes. Thus we join all common nodes $B \in \mathcal{N}_1 \cap \mathcal{N}_2$. On the automata level the join operator (denoted $\bowtie$) is performed as shown in Definition 2.2.

Definition 2.2 *(Product automaton)* The product automaton for the two constraint automata $\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \rightarrow_1, Q_{0,1})$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \rightarrow_2, Q_{0,2})$ is $\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \rightarrow, Q_{0,1} \times Q_{0,2})$ were $\rightarrow$ is defined by the following rules (let

[3] One cannot take from the constraint automata whether this is a write or take operation but in the Reo circuit message directions are shown

$q_1, p_1 \in Q_1$ and $q_2, p_2 \in Q_2$):

$$\frac{q_1 \xrightarrow{N_1,g_1}_1 p_1,\, q_2 \xrightarrow{N_2,g_2}_2 p_2,\, N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q_1, q_2\rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2\rangle} \tag{1}$$

$$\frac{q_1 \xrightarrow{N,g}_1 p_1,\, N \cap \mathcal{N}_2 = \emptyset}{\langle q_1, q_2\rangle \xrightarrow{N,g} \langle p_1, q_2\rangle} \tag{2}$$

And a third rule symmetric to the second one.

The definition of product automaton reflects the possible synchronous and asynchronous behavior of two joined constraint automata. The first rule (1) describes the synchronous behavior. After joining both automata only a synchronous step is possible for transitions involving common nodes. The data transferred in this step has to match $g_1 \wedge g_2$. The second rule (2) and the symmetric one define asynchronous actions. If $N \cap \mathcal{N}_2 = \emptyset$ the first automaton can make a step and the second one remains in the current state.

The second operation on Reo circuits also has a counterpart on the constraint automata level. Hiding a node in a constraint automaton produces a new automaton in which data at the hidden node is no longer observable from the outside. For a transition $q_1 \xrightarrow{\{A\}, d_A = d} q_2$ which only relies on the hidden port A this means that the transition can always be taken in state q_1.

Definition 2.3 *(Hiding on constraint automata)* Let $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, Q_0)$ be a constraint automaton and $C \in \mathcal{N}$. After hiding C the resulting constraint automaton $\exists C[\mathcal{A}] = (Q,\ \mathcal{N} \setminus \{C\},\ \rightarrow_C,\ Q_{0,C})$ is defined as follows. Let $\rightsquigarrow^*$ be the transition relation such that $q \rightsquigarrow^* p$ iff there exists a finite path

$$q \xrightarrow{\{C\}, g_1} q_1 \xrightarrow{\{C\}, g_2} q_2 \xrightarrow{\{C\}, g_3} \ldots \xrightarrow{\{C\}, g_n} p$$

where $g_1, \ldots, g_n$ only depend on C and are satisfiable. The set $Q_{0,C}$ of initial states is $Q_{0,C} = Q_0 \cup \{p \in Q : q_0 \rightsquigarrow^* p \text{ for some } q_0 \in Q_0\}$.

The transition relation $\rightarrow_C$ is given by

$$\frac{q \rightsquigarrow^* p,\ p \xrightarrow{N,g} e,\ \bar{N} = N \setminus \{C\} \neq \emptyset,\ \bar{g} = \exists C[g]}{q \xrightarrow{\bar{N},\bar{g}}_C e}$$

where $\exists C[g] = \bigvee_{d \in Data} g[d_C/d]$. We write $g[d_C/d]$ to denote the data constraint obtained by syntactically replacing all occurrences of d_C in g with d. To be more precise we replace all atoms $d_C = \bar{d}$ with $true$ if $d = \bar{d}$ and with $false$ if $d \neq \bar{d}$.

When constructing larger Reo circuits the question whether two circuits show the same observable behavior arise naturally. Also often it is easy to give a constraint automaton describing the desired behavior. Then comparing its behavior with the behavior of the constructed Reo circuit shows if the Reo circuit is a correct implementation. This is sometimes called a homogeneous approach to verification (model checking). There is a strong connection between this problem and the bisimilarity of constraint automata. An algorithm computing the bisimulation relation therefore can be of great help when treating larger Reo circuits. We start with some notations which are needed to introduce bisimulation for constraint automata.

For a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, Q_0)$, $q \in Q$, $N \subseteq \mathcal{N}$ and $P \subseteq Q$ we define $dc_{\mathcal{A}}(q, N, P) = \bigvee \{q : q \xrightarrow{N,g} p \text{ for some } p \in P\}$

If the automaton is clear from the context we leave out the subscript and write $dc(q, N, P)$. Then $dc(N, P) = \bigvee_{q \in Q} dc(q, N, P)$ $dc(q, N, P)$ stands for the weakest transition using only N ports from q to an arbitrary chosen state in P. If no such transition exists $dc(q, N, P) = false$.

With this notations we can now define bisimulation for constraint automata.

Definition 2.4 (*Bisimulation*) Let $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ be a constraint automaton. An equivalence relation $\mathcal{R}$ on Q is called bisimulation for $\mathcal{A}$ if for all pairs $(q_1, q_2) \in \mathcal{R}$, all $\mathcal{R}$-equivalence classes $P \in Q/\mathcal{R}$ and every $N \subseteq \mathcal{N}$, $dc(q_1, N, P) \equiv dc(q_2, N, P)$ holds.

Two states are called bisimilar (or bisimulation-equivalent) iff there exists a bisimulation $\mathcal{R}$ with $(q_1, q_2) \in \mathcal{R}$. Intuitively bisimilar states are equal powerful with respect to their outgoing transitions.

We write $q_1 \sim q_2$ iff q_1 and q_2 are bisimilar and $q_1 \nsim q_2$ iff they are not.

Two constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with the same set of names are bisimilar ($\mathcal{A}_1 \sim \mathcal{A}_2$) if in the bisimulation equivalence classes for the disjoint union automaton $\mathcal{B} = \mathcal{A}_1 \uplus \mathcal{A}_2$ for every initial state of $\mathcal{A}_1$ there exists a bisimilar initial state in $\mathcal{A}_2$ and vice versa.

3 Symbolic Constraint Automata

The basis of our implementation is a definition of a symbolic representation based on switching functions [7] that supports an efficient treatment of join and product. For our implementation we then use ordered binary decision diagrams [6][9][10][11] to represent and manipulate those switching functions. Let $\mathcal{Z} = \{z_1, \ldots, z_n\}$ be a finite set of boolean variables. An *evaluation* of $\mathcal{Z}$ is a function $\eta : \mathcal{Z} \rightarrow \{0, 1\}$ that assigns a value $\eta(z) \in \{0, 1\}$ to each variable $z \in \mathcal{Z}$. Eval($\mathcal{Z}$) identifies the set of all evaluations of $\mathcal{Z}$. Let $\overline{a} = (a_1, \ldots, a_n) \in \{0, 1\}^n$ and $\overline{z} = (z_{i_1}, \ldots, z_{i_n}) \in \mathcal{Z}^n$ such that $z_{i_j}, \ldots, z_{i_j}$ are distinct. Then $[\overline{z} = \overline{a}]$ denotes the evaluation $\eta \in \text{Eval}(\mathcal{Z})$ with $\eta(z_{i_j}) = a_j$, $j = 1, \ldots, n$. If $\overline{b} = (b_1, \ldots, b_r) \in \{0, 1\}^r$ and $\overline{z} = (z_{i_1}, \ldots, z_{i_r}) \in \mathcal{Z}^r$ with pairwise different variables z_{i_j} the evaluation $\eta \left[\overline{z} = \overline{b}\right] \in \text{Eval}(\mathcal{Z})$ agrees with η on all variables and assigns b_j to z for $z \in \{z_{i_1}, \ldots, z_{i_r}\}$ A *switching function* is a function $f : \text{Eval}(\mathcal{Z}) \rightarrow \{0, 1\}$. The set of all switching functions will be called $\mathbb{B}(\mathcal{Z})$ or $\mathbb{B}(z_1, \ldots, z_n)$. In the following we will use the common notions to denote operations on switching functions. Logical connectors like $\wedge$, $\neg$ and the derived operators for propositional logic have analog meanings for switching functions. To keep notations simple we use set operations like $\cap$ and $\cup$ with analog meaning for variable tuples like $\overline{b}$. Let $\bar{z}$ and $\overline{b}$ be as before and $f \in \mathbb{B}(\mathcal{Z})$. The *cofactor* of f related to $\bar{z}$ is defined by $f|_{\bar{z}=\bar{b}} \in \mathbb{B}(\mathcal{Z})$ where $f|_{\bar{z}=\bar{b}}(\eta) = f(\eta \left[\bar{z} = \overline{b}\right])$. Let $f \in \mathbb{B}(\mathcal{Z})$ and $z \in \mathcal{Z}$ then $\exists z[f] \in \mathbb{B}(\mathcal{Z})$ is given by $\exists z[f] = f|_{z=0} \vee f|_{z=1}$ To keep notations simple we write $\exists \bar{z}[f]$ as short form for $\exists z_1 \ldots \exists z_n[f]$ with $\bar{z} = (z_1, \ldots, z_n)$. Further we use $f(z' \leftarrow z)$ to denote a new function which is derived from f be replacing all z variables by z' variables.

For every constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, Q_0)$ we can now define the corresponding symbolic representation $(\delta(\bar{x}, \bar{y}, \overline{N}, \overline{d}), \chi_{init}(\bar{x}))$.

After renaming we can assume a state set $Q = \{q_0, \ldots, q_n\}, n \geq 2$ and we identify element q_i with its number i. Using $k = \lceil \log(n+1) \rceil$ boolean variables $\bar{x} = (x_1, \ldots, x_k)$ we define $\chi_{q_i}(\bar{x}) = x_1^{b_1} \wedge \ldots \wedge x_k^{b_k} \in (\mathbb{B}^k \rightarrow \mathbb{B})$ as symbolic representation for state q_i where $b_j \in \{0,1\},\ x_j^0 = \neg x_j,\ x_j^1 = x_j,\ j = 1, \ldots, k$ and $i = \sum_{j=1}^{k} b_j \cdot 2^j$.

As we want to reason about transitions from one state to another we have to distinguish between start and end states. We introduce copies of the x-variables (that serve to encode the start state of transitions) and deal with variables $\bar{y} = (y_1, \ldots, y_k)$ to represent the end states of transitions.

With $\overline{N} = (A_1, \ldots, A_m)$ we use the names as boolean variables for our ports. Further we define boolean variables $\overline{d} = (d_{A_1}, \ldots, d_{A_m})$ to represent the data items observed at the corresponding ports.

For every name set $N \subseteq \mathcal{N}$ we define

$$\chi_N(\overline{N}) = \bigwedge_{A_i \in N} A_i \ \wedge \bigwedge_{A_i \in \mathcal{N} \setminus N} \neg A_i$$

As we restricted our data domain to $\{0,1\}$ data constraints can be viewed as switching functions over $\overline{d}$.

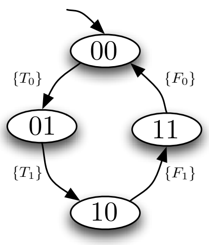


Fig. 2. Constraint automata for one PHIL component

Definition 3.1 *(Symbolic Constraint Automata)* A *symbolic constraint automaton* for a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, Q_0)$ is a tuple $\mathcal{A}_{sym} = (\delta(\bar{x}, \bar{y}, \overline{N}, \overline{d}), \chi_{init}(\bar{x}))$ where

$$\delta(\bar{x}, \bar{y}, \overline{N}, \overline{d}) = \bigvee_{(q,N,g,p) \in \longrightarrow} \chi_q(\bar{x}) \wedge \chi_N(\overline{N}) \wedge \chi_g(\overline{da}) \wedge \chi_p(\bar{y})$$

and $\chi_{init}(\bar{x}) = \bigvee_{q \in Q_0} \chi_q(\bar{x})$.

Example 3.2 Figure 2 shows the constraint automata that was introduced in Figure 1 after renaming the state set. The corresponding switching function is given by:

$$\begin{aligned}\delta(\bar{x},\bar{y},\overline{N},\overline{d}) = {} & (\neg x_1 \wedge \neg x_0 \wedge \neg y_1 \wedge y_0 \wedge T_0 \wedge \neg T_1 \wedge \neg F_0 \wedge \neg F_1) \\ & \vee (\neg x_1 \wedge x_0 \wedge y_1 \wedge \neg y_0 \wedge T_1 \wedge \neg T_0 \wedge \neg F_0 \wedge \neg F_1) \\ & \vee (x_1 \wedge \neg x_0 \wedge y_1 \wedge y_0 \wedge F_1 \wedge \neg T_0 \wedge \neg T_1 \wedge \neg F_0) \\ & \vee (x_1 \wedge x_0 \wedge \neg y_1 \wedge \neg y_0 \wedge F_0 \wedge \neg T_0 \wedge \neg T_1 \wedge \neg F_1)\end{aligned}$$

As in [2], the join operator is assumed to be preceded by an appropriate node-renaming such that all ports to be joined are named the same in both automata and therefore the variables for those ports and their constraints agree. Note that if A is a common node then A belongs the variable set of the symbolic representations of both automata.

Definition 3.3 *(Symbolic Product Automaton)* Given two symbolic constraint automata $\mathcal{A}_{sym} = (\delta_{\mathcal{A}}(\bar{x}_{\mathcal{A}},\bar{y}_{\mathcal{A}},\overline{N}_{\mathcal{A}},\overline{d}_{\mathcal{A}}),\chi_{init_{\mathcal{A}}}(\bar{x}_{\mathcal{A}}))$ and $\mathcal{B}_{sym} = (\delta_{\mathcal{B}}(\bar{x}_{\mathcal{B}},\bar{y}_{\mathcal{B}},\overline{N}_{\mathcal{B}},\overline{d}_{\mathcal{B}}),\chi_{init_{\mathcal{B}}}(\bar{x}_{\mathcal{B}}))$ with $\bar{x}_{\mathcal{A}} \cap \bar{x}_{\mathcal{B}} = \emptyset$ and $\bar{y}_{\mathcal{A}} \cap \bar{y}_{\mathcal{B}} = \emptyset$ the product automaton $\mathcal{C}_{sym} = \mathcal{A}_{sym} \bowtie \mathcal{B}_{sym}$ is defined as $\mathcal{C}_{sym} = (\delta_{\mathcal{C}}(\bar{x}_{\mathcal{C}},\bar{y}_{\mathcal{C}},\overline{N}_{\mathcal{C}},\overline{d}_{\mathcal{C}}),$ $\chi_{init_{\mathcal{C}}}(\bar{x}_{\mathcal{C}}))$ with $\bar{x}_{\mathcal{C}} = \bar{x}_{\mathcal{A}} \cup \bar{x}_{\mathcal{B}}$, $\bar{y}_{\mathcal{C}} = \bar{y}_{\mathcal{A}} \cup \bar{y}_{\mathcal{B}}$, $\delta_{\mathcal{C}}(\bar{x}_{\mathcal{C}},\bar{y}_{\mathcal{C}},\overline{N}_{\mathcal{C}},\overline{d}_{\mathcal{C}}) = \delta_{sync} \vee \delta_{async}$ and $\chi_{init_{\mathcal{C}}}(\bar{x}_{\mathcal{C}}) = \chi_{init_{\mathcal{A}}}(\bar{x}_{\mathcal{A}}) \wedge \chi_{init_{\mathcal{B}}}(\bar{x}_{\mathcal{B}})$. Where $\delta_{sync} = \delta_{\mathcal{A}} \wedge \delta_{\mathcal{B}}$ and

$$\delta_{async} = \left(\delta_{\mathcal{A}} \wedge \bigwedge_{A \in \mathcal{N}_{\mathcal{B}}} \neg A \;\wedge (\bar{x}_{\mathcal{B}} \leftrightarrow \bar{y}_{\mathcal{B}})\right) \vee \left(\delta_{\mathcal{B}} \wedge \bigwedge_{A \in \mathcal{N}_{\mathcal{A}}} \neg A \;\wedge (\bar{x}_{\mathcal{A}} \leftrightarrow \bar{y}_{\mathcal{A}})\right)$$

Where $\bar{x} \leftrightarrow \bar{y}$ stands for $\bigwedge_{i \in \{1,\ldots,k\}} (x_i \leftrightarrow y_i)$.

This operation on switching functions is the exact counterpart of our product automata operation defined in Definition 2.2. Every fullfilling evaluation for δ_{sync} is a fullfilling evaluation for $\delta_{\mathcal{A}}$ and $\delta_{\mathcal{B}}$. Thus the evaluation for $\overline{N}$ is such that common ports of both automata are equally active or passive. The conjunctive combination of the constraints g belonging to the transitions is given by the conjunctive combination of $\delta_{\mathcal{A}}$ and $\delta_{\mathcal{B}}$.

We obtain the asynchronous transitions by evaluating the δ_{async} function. The intuitive interpretation for the formula is $\mathcal{A}_{sym}$ makes a step which does not depend on ports in $\mathcal{B}_{sym}$. The formulas $(\bar{x}_{\mathcal{B}} \leftrightarrow \bar{y}_{\mathcal{B}})$ ensure that $\mathcal{B}_{sym}$ cannot make a step. This has to be done again with $\mathcal{A}_{sym}$ and $\mathcal{B}_{sym}$ switching their positions. By $\delta_{sync} \vee \delta_{async}$ we then compute the union of the synchronous and asynchronous transitions. Thus we obtain a function representing all transitions which exist in the product automaton.

Like the product operation hiding can also be done on the symbolic representation [4]. Because hiding relies on at least two steps we need another set of state variables $\bar{z}$. Remember the definition of hiding on constraint automata. In the symbolic case we have to perform a fixpoint iteration in order to collapse multiple transitions that rely only on the port we want to hide. A pseudo code version of this algorithm is shown in Algorithm 1. Our implementation also provides another variant of hiding using a repeated squaring approach. It depends on the example which one of them is slightly faster. Let C be the port we want to hide. First

[4] Hiding some nodes in our examples showed that you cannot expect the BDD representation of an automaton to be smaller after hiding.

we divide our system in two parts. δ_P contains all transitions which are labeled only with $\{C\}$. Analogously δ_R represents all transition which are not in δ_P. The first loop then computes our new set of initial states by performing a reachability analysis from all initial states using transitions in δ_P only. The second loop takes every two step transition $q \xrightarrow{\{C\}} p \xrightarrow{N,g} s$ and adds a new transition $q \xrightarrow{N,g} s$. If there is a sequence of consecutive transitions like $l \xrightarrow{\{C\}} q \xrightarrow{\{C\}} p \xrightarrow{N,g} s$ we first add $q \xrightarrow{N,g} s$. The next iteration then adds $l \xrightarrow{N,g} s$. The loop is continued until no new transition can be added. The last operation $\delta = \delta_R \wedge \neg\delta_P$ deletes all transitions labeled with $\{C\}$.

Algorithm 1: Hiding for symbolic constraint automata

```
$\delta_P = \delta \wedge C \wedge \bigwedge_{A \in \mathcal{N} \setminus \{C\}} \neg A$ ;        // all transitions labeled $\{C\}$
$\delta_R = \exists C \exists d_C [\delta \wedge \neg\delta_P]$ ;        // all other transitions
$\delta_P = \exists \overline{N} \exists \overline{d} [\delta_P]$ ;        // remove labels
repeat        // compute new initial states
    $\chi_{old} := \chi_{init}$;
    $\chi_{init} := \chi_{init} \vee \exists \bar{x}(\chi_{init} \wedge \delta_P)\{\bar{x} \leftarrow \bar{y}\}$;
until $\chi_{init} = \chi_{old}$;
repeat        // for $q \xrightarrow{\{C\}} p \xrightarrow{N,g} s$ add new transitions $q \xrightarrow{N,g} s$
    $\delta_O := \delta_R$;
    $\delta_R := \delta_P \wedge \delta_R\{\bar{z} \leftarrow \bar{y}, \bar{y} \leftarrow \bar{x}\}$;
    $\delta_R := \exists \bar{y}[\delta_R]\{\bar{y} \leftarrow \bar{z}\}$;
    $\delta_R := \delta_R \vee \delta_O$;
until $\delta_O = \delta_R$;
$\delta = \delta_R \wedge \neg\delta_P$ ;        // delete all $\{C\}$-transitions
return $(\delta(\bar{x}, \bar{y}, \overline{N} \setminus \{C\},\ \overline{d} \setminus \{d_C\}),\ \chi_{init}(\bar{x}))$
```

When reasoning about equivalence using bisimulation we need to regard two disjoint automata $\mathcal{A}$ and $\mathcal{B}$ as one big automaton. We call the resulting automaton $\mathcal{C}$ a disjoint union of two automata denoted by $\mathcal{C} = \mathcal{A} \uplus \mathcal{B}$. For explicit representations with disjoint node sets the node set of the new automaton is the union of both node sets. The new transition relation is the union of both transition relations. The set of initial states consists of all states that are in the union of the two sets of initial states of the disjoint automata. The same operation on symbolic representations can be done but we have to pay attention to non-essential variables when putting the two automata together.

Symbolic representation using switching functions to encode transition relations and binary decision diagrams to represent those switching functions is a powerful method to handle large transition systems [7].

To obtain compact BDD-representations of constraint automata the variable ordering must be chosen carefully. A detailed description of our heuristics to determine a good variable ordering goes beyond the scope of this paper. We just mention that our heuristics attempt to put variables representing constraints close to variables representing states having transitions relying on those constraints.

4 Bisimulation

Checking whether two automata are bisimilar ($\mathcal{A}_1 \sim \mathcal{A}_2$) is known to be coNP-hard [2]. Therefore we cannot expect good performance for every instance when computing bisimulation quotients. A well known way to compute bisimulation quotient for ordinary labeled transition systems is the algorithm by Kannelakis/Smolka [8]. A sketch of how to adapt this algorithm for constraint automata is presented in [2]. We start with some notation needed for this algorithm.

Partition: For a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, Q_0)$ a partition for Q stands for a set $\Pi = \{P_1, \ldots, P_n\}$ of pairwise disjoint, non-empty subsets of Q such that $Q = P_1 \cup \ldots \cup P_n$. For a partition $\Pi = \{P_1, \ldots, P_n\}$ of Q we call the elements P_i blocks. A super-block then denotes a non-empty union of blocks in Π. A splitter denotes a pair (N, P) consisting of a non-empty subset N of $\mathcal{N}$ and a super-block P for Π.

Refine: Let Π be a partition for Q, (N, P) a splitter for Π and B a block for Q. For two states q_1,q_2 we then define the equivalence $\equiv_{(N,P)}$ such that

$$q_1 \equiv_{(N,P)} q_2 \text{ iff } dc(q_1, N, P) = dc(q_2, N, P)$$

Then we define $Refine(B, N, C) = B/\equiv_{(N,P)}$. A block B is called stable with respect to (N, P) if $Refine(B, N, P) = \{B\}$. For a refinement of the whole partition we write $Refine(\Pi, N, P) = \bigcup_{B \in \Pi} Refine(B, N, P)$

The idea of the partitioning splitter algorithm is to construct a sequence $\Pi_0, \Pi_1, \ldots, \Pi_k$ of partitions with Π_{i+1} is strictly finer than Π_i but coarser than the bisimulation quotient $Q/\sim$. For a finite set Q we get $\Pi_k = Q/\sim$ for some $k \leq |Q|$. A brief sketch of this algorithm is shown in Algorithm 2. The most critical part is the calculation of the logical equivalence classes of the data constraints and the refinement of the actual block according to those classes. This operation is performed on our symbolic representation using the following ideas. First we choose an arbitrary state q out of B. Then we select its transitions to the current splitter. The function describing the labeling of this transition is the pattern function p for the pattern equivalence operator. With $\mathcal{Z}' = \overline{N} \cup \overline{d}$ we then obtain a function representing all states that are in the same refined block as q. We remove these states from the current block and start with a new arbitrarily chosen state if one is left.

We implemented a symbolic version of Algorithm 2 based on ordered binary decision diagrams (OBDDs) in order to estimate its efficiency for constraint automata. Note that our approach is not restricted to using OBDDs. For this partitioning splitter algorithm we have to compute logical equivalence classes with respect to the data constraints. This cannot be done by the standard operators provided by an OBDD library. The main idea of our approach is to arbitrarily choose a state and regard its outgoing transitions as representative for one equivalence class. What remains is to efficiently compute the class belonging to this representative. For this purpose we define a special pattern equivalence operator on switching functions in order to find logical equivalence classes. First we start with a function getAssignment($\check{f}$) which chooses an arbitrary fullfilling evaluation for $\check{f}$ and returns a switching function that

Algorithm 2: Partitioning splitter algorithm

```
$\Pi := Q$;
$Splitters := \{(N,Q) : N \subseteq \mathcal{N}, \bigvee_{q \in Q} dc(q,N,Q) \not\equiv \text{false}\}$ ;
while $Splitters \neq \emptyset$ do
    choose$(N,P) \in Splitters$ and remove $(N,P)$ from $Splitters$;
    forall $B \in \Pi$ do
        find equivalence classes $D_1, \dots, D_r$ of $dc(q,N,P),\ q \in B$;
                              // If $r = 1$ then $B$ is stable w.r.t. $(N,P)$
                                 // in other words Refine$(B,N,P) := \{B\}$
        if $r \geq 2$ then                                  // $B$ is not stable
            $B_i = \{q \in Q : dc(q,N,P) \in D_i\}$;
                                       // Refine$(B,N,P) = \{B_1, \dots, B_r\}$
            $\Pi := (\Pi \setminus \{B\}) \cup \{B_1, \dots, B_r\}$;
            forall $(\tilde{N}, B_i)$ with $\emptyset \neq \tilde{N} \subseteq \mathcal{N},\ dc(\tilde{N}, B_i) \not\equiv false$ do
                add $(\tilde{N}, B_i)$ to $Splitters$;
return $\Pi$;
```

is true for this evaluation and false for all other evaluations.[5] This function can be used to select one state and its transitions. Using an existential quantification we then compute the representative for the logical equivalence class. It remains to explain how to compute the switching function describing the set of all states in this logical equivalence class. This can be done by applying our pattern equivalence operator on the state space representation with function p being our representative and therefore $\mathcal{Z}'$ consisting of all variables representing labels of transitions.

Definition 4.1 Let $\mathcal{Z} = \{z_1, \dots, z_n\}$ be a finite set of boolean variables and $\mathcal{Z}' = \{z'_1, \dots, z'_k\} \subset \mathcal{Z}$ be a subset of $\mathcal{Z}$. Let $f \in \mathbb{B}(\mathcal{Z})$ a function and $p \in \mathbb{B}(\mathcal{Z}')$ a pattern function. The pattern equivalence of f with respect to p (written $f/_{\equiv_p}$) is defined by: $f/_{\equiv_p} : Eval(\mathcal{Z} \backslash \mathcal{Z}') \to \mathbb{B}$ where

$$f/_{\equiv_p}(\eta) = \begin{cases} 1 & p = f|_\eta \text{ where } f|_\eta \text{ is the cofactor of } f \text{ with respect to } \eta \\ 0 & \text{otherwise} \end{cases}$$

with $\eta \in Eval(\mathcal{Z} \backslash \mathcal{Z}')$.

□

Example 4.2 Let $\mathcal{Z} = \{x, y, z\}$ and $\mathcal{Z}' = \{y\}$. The function $f(x,y,z) = (x \wedge y) \vee z$ and the pattern function $p(y) = y$. The results for $f/_{\equiv_p}$ are shown in the following tables.

[5] At the OBDD level this means a representative for a path to the 1-sink

x, z	$f(x, z)$
00	0
01	1
10	y
11	1

$\Rightarrow$

x, z	$f/_{\equiv_p}(x, z)$
00	0
01	0
10	1
11	0

The pattern equivalence operator allows to efficiently handle symbolic representations of the whole automata in contrast to [12] that treads small state spaces explicitly and only uses symbolic methods for the labeling of transitions. In contrast to [13] our approach is capable of handling the rich labeled transitions of constraint automata directly. As our algorithm works on a symbolic representation a complexity analysis is leads to distorted results. One can count the number of symbolic operations but the OBDDs this operations work on can have exponential size in the number of variables.

5 Benchmarks

After describing how the partition splitter algorithm can be implemented on symbolic constraint automata we now present some benchmarks for the implementation of this algorithm.

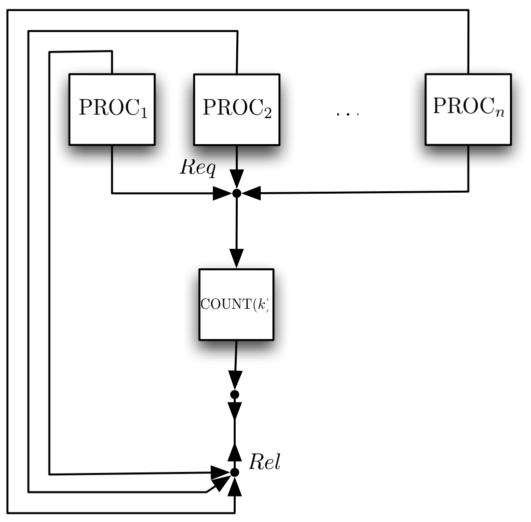
In Reo a FIFO(N)-component should behave like N FIFO(1)-components after joining them together and hiding away the inner nodes. We used another approach and tried to specify the constraint automaton for a FIFO(N) component directly. Using the partitioning splitter algorithm we found very subtle differences in these two implementations. When building the FIFO(N) directly we considered two possible operations. One can put one data element into the FIFO if the FIFO is not full and one can take one out of it when it is not empty. In contrast using the Reo construction leads to a component which can perform both operations at the same time. Our constraint automata does not specify the behavior of a correct Reo FIFO(N)-component. So the two approaches lead to non-bisimular constraint automata. To gain bisimilar automata we can eliminate the concurrent operations by adding an AsyncDrain channel which prevents the incoming and outgoing ports from performing operations at the same time. Benchmark results for showing bisimulation equivalence of this components are shown in Tab. 1. Note that the automaton resulting from direct construction has much fewer states than the automaton built from the Reo circuit gained through join and hide. All states of the first automaton build their own equivalence class. While the states of the second automaton are divided in exactly the same number of equivalent classes. This example shows how join and hide can lead to redundancy in the resulting constraint automaton.

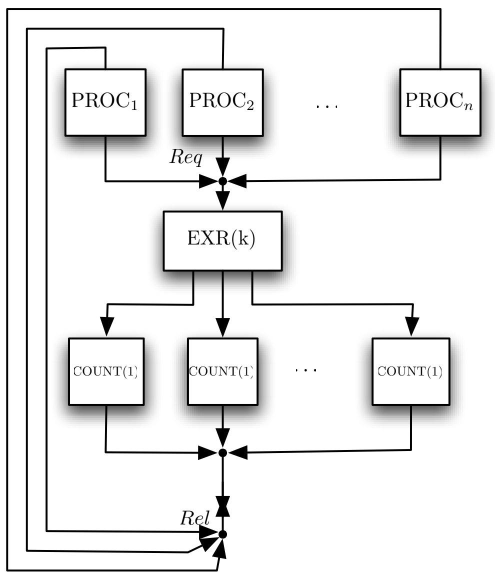
n	$Memory$	$states_1$	$trans_1$	$states_2$	$trans_2$	$splitters$	$classes$	$Time_{bisim}$
6	69	127	252	567	7140	376	127	1.87
7	73	255	508	1701	35566	760	255	4.22
8	80	511	1020	5103	179508	1528	511	17.57
9	95	1023	2044	15309	916574	3064	1023	75.94
10	-	-	-	-	-	-	-	time out

Table 1
Bisimulation and FIFO(N) versus n·FIFO(1)

n	k	$Memory$	$states$	$transitions$	$splitters$	$classes$	$Time_{bisim}$
200	5	81	$5.47 \cdot 10^{9}$	$1.62 \cdot 10^{11}$	20	6	0.52
200	20	103	$1.37 \cdot 10^{28}$	$7.98 \cdot 10^{30}$	80	21	2.93
200	30	120	$3.76 \cdot 10^{37}$	$5.07 \cdot 10^{40}$	120	31	6.95
200	60	217	$5.84 \cdot 10^{59}$	$2.50 \cdot 10^{63}$	240	61	43.64
400	5	105	$1.37 \cdot 10^{11}$	$5.17 \cdot 10^{12}$	20	6	1.48
400	20	145	$1.05 \cdot 10^{34}$	$5.63 \cdot 10^{36}$	80	21	6.95
400	30	172	$1.50 \cdot 10^{46}$	$4.23 \cdot 10^{49}$	120	31	12.39
400	60	271	$1.25 \cdot 10^{76}$	$8.82 \cdot 10^{79}$	240	61	71.75
800	5	174	$5.50 \cdot 10^{12}$	$1.65 \cdot 10^{14}$	20	6	4.57
800	20	251	$9.98 \cdot 10^{39}$	$4.84 \cdot 10^{42}$	80	21	19.81
800	30	304	$1.11 \cdot 10^{58}$	$1.40 \cdot 10^{58}$	120	31	32.80
800	60	388	$1.74 \cdot 10^{93}$	$1.07 \cdot 10^{97}$	240	61	148.92

Table 2
Bisimulation and the mutual exclusion protocol (PROC2)





Another example used for our benchmarks are the two different Reo circuits specifying a mutual exclusion protocol shown above. This protocol specifies the interaction of n processes that want to enter a critical section of their execution while only k processes are allowed to be in the critical section at the same time. Using the algorithm presented in this paper we could prove that the two versions do not to lead to the same observable behavior. This is because the left one allows one process to enter its critical section while another one leaves its critical section at the same time. One can add an asynchronous drain channel (which accepts all data at its ends but not at both at the same time) connecting nodes *Req* and *Rel* to render this impossible. Then the constraint automata for the two circuits become bisimilar. Results for this example are shown in Tab. 2.

6 Conclusion and Perspective

Using Reo to model the behavior of coordination protocols raises the question whether two connectors have the same observable behavior. This shows the need for a tool to answer this question in an automated way. This work shows how techniques known to work for labeled transition systems can be adapted to constraint automata. Introducing a symbolic representation for constraint automata and explaining how the product and hide operator can be performed in a symbolic way. Large Reo networks with many components can by handled automatically. In order to compare two constraint automata we show how a partition splitter algorithm for computing the bisimulation equivalence classes of constraint automata can be adapted to work on our symbolic representation. By introducing the pattern equivalence operator for switching functions we gain a powerful tool for symbolic treatment of the rich labeled transitions that occur in constraint automata. The benchmarks section gives an impression of how fast constraint automata for huge Reo circuits can be constructed and compared using our ideas. Future work will address several improvements of our implementation. Especially the set of splitters may be implemented in a more efficient way. At the moment we are working on combining our tool with the BTSL model checker for constraint automata presented in [14] and integrating both with the graphical design tool for Reo circuit currently being developed at CWI Amsterdam. Furthermore bisimulation as a homogenous approach to model checking constraint automata should be extended to richer versions of constraint automata like timed or probabilistic ones. Also one should extend the ideas presented here to make them applicable for constraint automata with priorities.

7 Acknowledgment

We like to thank Jörn Ossowski for his help implementing the pattern equivalence operator for his OBDD library.

References

[1] F. Arbab, Reo: A Channel-based Coordination Model for Component Composition, Mathematical Structures in Computer Science, 14(3):1-38, 2004

[2] C. Baier and M. Sirjani and F. Arbab and J.J.M.M. Rutten, Modeling Component Connectors in Reo by Constraint Automata, Science of Computer Programming, 61:75-113, 2006

[3] CAPD: Construction and Analysis of Distributed Processes, http://www.inrialpes.fr/vasy/cadp/

[4] CWB: The Edinburgh Concurrency Workbench, http://homepages.inf.ed.ac.uk/perdita/cwb/

[5] F. Arbab, Abstract Behavior Types: A Foundation Model for Components and Their Composition, Lecture Notes in Computer Science Volume 2852/2003, pp 33-70

[6] R. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Transactions on Computers, C-35, 1986

[7] K. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993

[8] P. Kannelakis and S. Smolka. Finite State Processes and three Problems of Equivalence. In CCS expressions, pages 228-240, 1983. Proc. 2nd ACM Symposium on the Principles of Distributed Computing 1983

[9] S. Minato, N. Ishiura, S. Yajima, Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation, Proceedings of the 27th ACM/IEEE Design Automation Conference, ACM, New York, pp 52-57, 1990

[10] C. Meinel and T. Theobald, Algorithms and Data Structures in VLSI Design, Springer-Verlag, 1998

[11] I. Wegener, Branching Programs and Binary Decision Diagrams. Theory and Applications, Monographs on Discrete Mathematics and Applications, SIAM, 2000

[12] Parosh Aziz Abdulla, Lisa Kaati, Marcus Nilsson, Minimization of Non-Deterministic Automata with Large Alphabets, Proc. CIAA'05, 2005

[13] Amar Bouali and Robert de Simone, Symbolic Bisimulation Minimisation, Computer Aided Verification, pp 96-108, 1992

[14] S. Klüppelholz and C. Baier, Symbolic Model Checking for Channel-based Component Connectors, FOCLASA 2006

[15] J. Ossowski, JINC, a bdd library, www.jossowski.de.