Symbolic Model Checking for Channel-based Component Connectors

Sascha Klüppelholz, Christel Baier

Universität Bonn, Institut für Informatik I, Germany
{klueppel,baier}@cs.uni-bonn.de

Abstract

The paper reports on the foundations and experimental results with a model checker for component connectors modelled by networks of channels in the calculus Reo. The specification formalisms is a branching time logic that allows to reason about the coordination principles of and the data flow in the network. The underlying model checking algorithm relies on variants of standard automata-based approaches and model checking for CTL-like logics. The implementation uses a symbolic representation of the network and the enabled I/O-operations by means of binary decision diagrams. It has been applied to a couple examples that illustrate the efficiency of our model checker.

Keywords: constraint automata, model checking, branching time logic, data streams, binary decision diagrams

1 Introduction

In the past 15 years, many languages and models for coordination have been developed that provide a formal description of the glue code for plugging components together and can also serve as a starting point for formal verification. In this paper, we address the latter aspect for the exogenous coordination language Reo [2]. In Reo, the glue code is provided by a network of channels obtained through a series of operations that create channel instances and link them together in (network) nodes. The semantics of Reo networks has been provided in different, but consistent ways. [2] formalizes the enabledness and effect of I/O-operations at the network configurations by means of accept and offer predicates that declare whether and which data items can be written or read at a node. An operational semantics that specifies the stepwise behavior of and possible data flow in a Reo network has been presented in [6] using a variant of labelled transition systems, called constraint automata, and shown to be consistent with the timed data stream semantics of [5].

Although Reo is an elegant formalism to synthesize component connectors with simple composition operators, Reo networks with many channels tend to be hard to understand. Thus, tool support for analyzing the coordination mechanism specified by a Reo network

[1] The authors are supported by the DFG-NWO-projects SYANCO and VOSS II.

is a crucial aspect for applying the Reo framework for complex scenarios. Algorithms for verifying Reo networks on the basis of their constraint automata semantics have been presented in [6] for checking (bi)simulation and language equivalence and in [3,10] for temporal logic specifications. We follow here the latter approach and deal with a branching time, time-abstract variant of timed data stream logic (TDSL) introduced in [3] for reasoning about real-time constraints of Reo networks in the linear time setting. Ignoring some minor differences, our logic, called branching time stream logic (*BTSL*), is contained in the logic considered in [10], where the main focus is on the treatment of dynamic reconfiguration rather than model checking. *BTSL* combines the standard CTL-operators [11,12] with a special path modality $\langle\alpha\rangle$ and its dual $[\alpha]$ that allow to reason about the data streams observable at the network nodes by means of a regular expression α. For instance, assume $\mathcal{C}$ is a component which is linked to a Reo network by an output port *Request* where $\mathcal{C}$ sends off the request to get access to certain resources and an input port *Grant* where $\mathcal{C}$ might receive the grant. Then, the *BTSL* formula

$$\exists[\mathit{true}^*\,;\mathit{Request}\,;(\neg \mathit{Grant})^*]\,\forall\langle \mathit{true}^*\,;\mathit{Grant}\rangle\,\mathit{resources\_available}$$

states the possibility that each request of $\mathcal{C}$ will eventually be granted and the required resources will be available for $\mathcal{C}$.

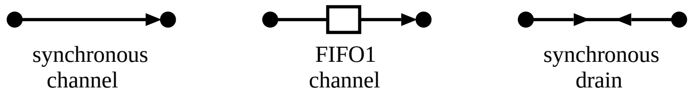
The purpose of this paper is to report on an implementation of a *BTSL* model checker. The input is a Reo network and a *BTSL* formula Φ which has to be checked for the network. The *BTSL* model checking procedure relies on a combination of known methods for model checking CTL-like logics and automata-based approaches for linear time logics. A rough sketch for model checking a *BTSL*-like logic has been given in [10], which follows the standard CTL* model checking approach [14,12] and uses a reduction to the TDSL model checking problem. However, no details or explanations on an efficient implementation have been provided in [10]. In fact, for *BTSL* the reduction to the model checking problem for the real-time logic TDSL is unnecessary, since simpler techniques sufficies. As we will show in this paper, for the treatment of the modalities $\langle\alpha\rangle$ and $[\alpha]$ even a reduction to ordinary CTL is possible. Furthermore, we depart from former approaches with constraint automata by dealing with infinite and finite runs. The latter are crucial for the treatment of deadlock configurations that might appear in Reo networks.

Our model checker deals with a symbolic approach where the constraint automaton for a Reo network is represented by a binary decision diagram (BDD). The first step is the generation of a BDD-representation of the contraint automaton for the network. This is done in a compositional manner by mimicking Reo's operators to synthesize the network by adding channels and joining nodes by means of corresponding operators on BDDs. The second step is then to perform the *BTSL* model checking using appropriate operations for manipulating BDDs. For this, we apply state-of-the-art techniques for symbolic CTL model checking in combination with a symbolic treatment of the $\langle\alpha\rangle$- and $[\alpha]$-modality.

Organisation of the paper. Section 2 gives a brief introduction in the coordination language Reo and constraint automata that serve as operational model for Reo networks. In Section 3, we explain the syntax and semantics of the logic *BTSL*. Section 4 summarizes the main steps of the *BTSL* model checking algorithm and reports on our symbolic implementation. Experimental results will be presented in Section 5. Section 6 concludes the paper.

2 Reo and constraint automata

In this section we summarize the main concepts of the coordination language Reo and its operational constraint automata semantics. Further details can be found in [2,6]. Reo is an exogeneous coordination language which is based on a channel-based calculus where complex component connectors are organized in a network of channels and built in a compositional manner. Reo networks provide the glue code for the coordination and interactions of the components that are connected to the network. Reo relies on a very liberal notion of channels and supports any kind of peer-to-peer communication. The requirement for the channels used in a Reo network are that channels must have two channel ends, declared to be sink or source ends, and a user-defined semantics. At source ends data items enter the channel (by performing corresponding write operations), while data items are received from a channel at the sink ends (by performing corresponding read operations).



The figure above shows the graphical representation of three simple channel types that will be used in our examples. Synchronous and FIFO channels have a source and a sink end. In synchronous channels the write and read operations have to be performed simultaneously. The picture in the middle shows a FIFO channel with one buffer cell, briefly called FIFO1 channel, where the buffer is initially empty. Writing a data item at the source end is enabled as long as the buffer is empty. The effect of writing d is that d will be stored in the buffer. Reading at the sink end is enabled if the buffer is filled, in which case the data item is taken off from the buffer. A very useful channel for the design of complex coordination principles in Reo is the synchronous drain. It has two source ends (but no sink end). A data item has to be written on both ends simultaneously and both are being destroyed.

The nodes of a Reo network represent sets of channel ends. They arise through Reo's join operator and can be classified into source, sink and mixed nodes, depending on whether all channel ends that coincide on a node A are source ends (then A is a source node), sink ends (then A is a sink node), or whether A combines sink and source ends (then A is a mixed node). Source and sink nodes represent input and output ports where components might connect to the network. The mixed nodes serve as routers where data items can be transmitted through the network.

Concurrent I/O-operations. For simplicity of the paper, we assume here a fixed finite and nonempty set *Data* of data items that can be written or taken from the channels. If $\mathcal{N}$ is a set of network nodes then the observable data flow at some moment can be described by a *concurrent I/O-operation*. This means a pair (N,δ) where N is a nonempty node-set (i.e., $\emptyset \neq N \subseteq \mathcal{N}$) and $\delta : N \to \mathit{Data}$. The intuitive meaning of a concurrent I/O-operation (N,δ) is that the nodes $A \in N$ synchronize their I/O-operations such that $\delta(A)$ is the data item observed at node A. More precisely, each source node $A \in N$ writes data item $\delta(A)$ at all channels with a source end on A, while each sink node $A \in N$ takes data item $\delta(A)$ from one of the channels with a sink end on A. The mixed nodes $A \in N$ read $\delta(A)$ from one of the channels with a sink end on A and simultaneously writes $\delta(A)$ at all channels with a source end on A. In the moment where the concurrent I/O-operation (N,δ) is performed there is no data flow at the other nodes $B \in \mathcal{N} \setminus N$.

Constraint automata have been introduced to provide a compositional, operational semantics for Reo networks [6]. The states of the automaton for a Reo network represent the configurations (e.g., contents of the buffers for FIFO channels), while the transitions model the enabled concurrent I/O-operations. In [6] the transitions have the form $q \xrightarrow{N,dc} p$ where q and p are the starting and target states, respectively, N is the set of nodes where I/O-operations are performed simultaneously and dc is a data constraint, i.e., a boolean condition on the data items written or read at the nodes $A \in N$. According to our BDD-based implementation (see Section 4), we go one step further towards a symbolic representation and deal with transitions $q \xrightarrow{g} p$ where g is an I/O-constraint, i.e., a condition on both, the nodes where I/O-operations will be performed and the data items. Furthermore we depart from [6] by dropping the requirement that all runs have to be infinite. We also deal with finite runs, which are necessary to argue about deadlock configurations.

I/O-constraints. We use a symbolic representation of *sets* of concurrent I/O-operations by means of boolean conditions on the nodes $A \in \mathcal{N}$ and the data items d_A written or read at node A. Formally, an *I/O-constraint* for $\mathcal{N}$ is a propositional formula built by the literals A (where $A \in \mathcal{N}$) and the atomic formulas "$(d_{A_1}, \ldots, d_{A_k}) \in R$" where $k \geq 1$, $A_1, \ldots, A_k$ are pairwise distinct nodes and $R \subseteq Data^k$. Throughout the paper, we will use intuitive notations like "$d_A = 0$" for "$d_A \in \{0\}$" or "$d_A \neq d_B$" for "$(d_A, d_B) \in \{(\delta_A, \delta_B) \in Data^2 \mid \delta_A \neq \delta_B\}$", and write *IOC* for the set of all I/O-constraints. I/O-constraints are interpreted over concurrent I/O-operations (N, δ) in the expected way, i.e., $(N, \delta) \models A$ iff $A \in N$ and $(N, \delta) \models (d_{A_1}, \ldots, d_{A_k}) \in R$ iff $\{A_1, \ldots, A_k\} \subseteq N \wedge (\delta(A_1), \ldots, \delta(A_k)) \in R$. The propositional logic operators have their standard semantics. We write $[\![g]\!]_{\mathcal{N}}$ for the set of concurrent I/O-operations (N, δ) where $\emptyset \neq N \subseteq \mathcal{N}$ and $(N, \delta) \models g$. Note that the semantics of I/O-constraints depends on the underlying node-set $\mathcal{N}$. For example, $[\![d_A = d_B]\!]_{\mathcal{N}} = \{(N, \delta) \mid \{A, B\} \subseteq N \subseteq \mathcal{N}, \delta(A) = \delta(B)\}$ and $[\![true]\!]_{\mathcal{N}} = \{(N, \delta) \mid \emptyset \neq N \subseteq \mathcal{N}, \delta : N \to Data\}$. Two I/O-constraints g_1 and g_2 are $\mathcal{N}$-equivalent, denoted $g_1 \equiv_{\mathcal{N}} g_2$, iff $[\![g_1]\!]_{\mathcal{N}} = [\![g_2]\!]_{\mathcal{N}}$. If the node-set is clear from the context we simply write $[\![g]\!]$ and $\equiv$ and speak about satisfiability and equivalence of I/O-constraints.

Definition 2.1 A constraint automaton (CA) is a tuple $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0, AP, L)$, where Q is a set of states, $\mathcal{N}$ a set of nodes, disjointly partioned into $\mathcal{N} = \mathcal{N}^{src} \uplus \mathcal{N}^{snk} \uplus \mathcal{N}^{mix}$, $Q_0 \subseteq Q$ the set of initial states, $\longrightarrow \subseteq Q \times IOC \times Q$ the transition relation, AP a finite set of atomic propositions, and $L : Q \to 2^{AP}$ a labeling function. The nodes in $\mathcal{N}^{src}$ ($\mathcal{N}^{snk}$, $\mathcal{N}^{mix}$) are called source nodes (sink nodes and mixed nodes, respectively). The instances of a transition (q, g, p) are tuples (q, N, δ, p) where $(N, \delta) \in [\![g]\!]_{\mathcal{N}}$. Throughout the paper, we only consider finite constraint automata, i.e., we require that $\mathcal{N}$, Q and $\longrightarrow$ are finite.

■

In the sequel, we use arrow-notations $q \xrightarrow{g} p$ for a transition (q, g, p) and $q \xrightarrow{N,\delta} p$ for its instances. Fig. 1 illustrates the constraint automata for a synchronous channel with source node A and sink node B, a FIFO1 channel with source node A and sink node B and the data domain $Data = \{0, 1\}$ and a synchronous drain with source nodes A and B. In all three cases the node set is $\mathcal{N} = \{A, B\}$. The I/O-constraint "$d_A = d_B$" in the automaton for the synchronous channel indicates the concurrent I/O-operations $(\{A, B\}, \delta)$ where $\delta(A) = \delta(B)$, while the I/O-constraint $A \wedge B$ in the automaton for the synchronous drain represents all concurrent I/O-operations of the form $(\{A, B\}, \delta)$. For the FIFO channel one might use the atomic propositions *empty* and *full* with the obvious labeling function.

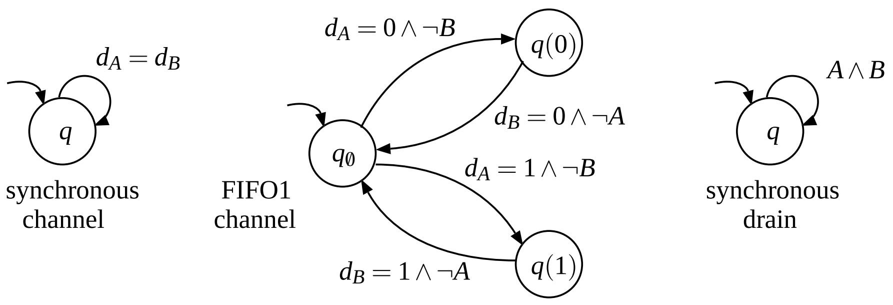


Fig. 1. CA for a synchronous channel, FIFO1 channel and synchronous drain

For state q, the I/O-constraints $ioc(q,p) = \bigvee\{g \mid q \xrightarrow{g} p\}$ represents the weakest condition on the I/O-operations at the nodes that have to be synchronized for moving within one step from q to p. Thus, if $P \subseteq Q$ then $ioc(q,P) = \bigvee_{p \in P} ioc(q,p)$ stands for the set of all concurrent I/O-operations that are enabled in q and lead to a configuration in P. With $P = Q$, we get a boolean characterization $ioc(q) = ioc(q,Q)$ for the set of all enabled concurrent I/O-operations in q.

State q is called *terminal* if $ioc(q) \wedge \bigwedge_{A \in \mathcal{N}^s} \neg A \equiv$ *false* where $\mathcal{N}^s = \mathcal{N}^{src} \cup \mathcal{N}^{snk}$. This condition means that in all enabled concurrent I/O-operations in q at least one of the sink or source nodes is involved. These I/O-operations might be refused if the components that connect to these nodes are not willing to provide the corresponding write or read operations. Thus, data flow might stop in terminal states.

The intuitive operational behavior of a constraint automaton can be formalized by its runs. Runs in a constraint automaton are defined as finite or infinite sequences of consecutive transition instances. In the case of finite runs, we allow that they end with a special pseudo-transition with the label $\surd$, denoting the end of data flow, provided that the last state is terminal. I.e., finite runs have the form

$$(1)\ \ q_0 \xrightarrow{N_1,\delta_1} \ldots \xrightarrow{N_k,\delta_k} q_k \quad \text{or} \quad (2)\ \ q_0 \xrightarrow{N_1,\delta_1} \ldots \xrightarrow{N_k,\delta_k} q_k \xrightarrow{\surd} q_k$$

where $q_{i-1} \xrightarrow{N_i,\delta_i} q_i$ are transition instances ($i = 1,\ldots,k$) and q_k is terminal for finite runs ending with a $\surd$-transition (case (2)). The length $|\theta| \in I\!N \cup \{\omega\}$ is defined as the number of transition instances taken in θ (possibly including the pseudo-transition with label $\surd$). A maximal run means an infinite run or a finite run that ends with a pseudo-transition labelled by $\surd$. We write $Runs(q)$ for the set of all runs starting in q and $MaxRuns(q)$ for all maximal runs starting in q.

If $\theta = q_0 \xrightarrow{N_1,\delta_1} q_1 \xrightarrow{N_2,\delta_2} q_2 \xrightarrow{N_3,\delta_3} \ldots$ is an infinite or finite, but non-maximal run then the word $(N_1,\delta_1)\,(N_2,\delta_2)\,(N_3,\delta_3)\ldots$ obtained by taking the projection to the sequence of concurrent I/O-operations is called the I/O-stream of θ. For finite maximal runs, say $\theta = q_0 \xrightarrow{N_1,\delta_1} \ldots \xrightarrow{N_k,\delta_k} q_k \xrightarrow{\surd} q_k$, the I/O-stream of θ is the word $(N_1,\delta_k)\ldots(N_k,\delta_k)\,\surd$.

3 Branching Time Stream Logic

In this section we introduce a branching time temporal logic for reasoning about the control and data flow of a constraint automata. The logic, called Branching Time Stream Logic (*BTSL*), combines features of CTL [11,12], PDL [15] and timed data stream logic (TDSL) [3,9,4]. As in CTL, formulas may refer to the configurations of a component connector (states of a constraint automata) by means of atomic propositions $ap \in AP$ and may use the path quantifiers $\exists$ and $\forall$. Path properties are specified by the standard until operator or the PDL/TSDL-like modality $\langle\alpha\rangle$ where α is a regular expression specifying sequences of I/O-operations at the nodes.

Branching Time Stream Logic (*BTSL*). A *BTSL* signature is a tuple $(AP, \mathcal{N})$ consisting of a finite nonempty set AP of atomic propositions and a finite nonempty node-set $\mathcal{N}$. The syntax of *BTSL* has three levels: state formulas (denoted by capitol greek letters Φ, Ψ), run formulas (denoted by the small greek letter φ), and regular I/O-stream expressions (denoted by the letter α). The abstract syntax of *BTSL* is given by the following grammar where $ap \in AP$ and $g \in IOC$:

$$\begin{aligned}
\Phi &:= \mathit{true} \;\big|\; ap \;\big|\; \Phi_1 \wedge \Phi_2 \;\big|\; \neg\Phi \;\big|\; \exists\varphi \;\big|\; \forall\varphi \\
\varphi &:= \Phi_1 \;\mathcal{U}\; \Phi_2 \;\big|\; \langle\alpha\rangle\Phi \\
\alpha &:= g \;\big|\; \mathit{stop} \;\big|\; \alpha^* \;\big|\; \neg\alpha \;\big|\; \alpha_1 ; \alpha_2 \;\big|\; \alpha_1 \cup \alpha_2 \;\big|\; \alpha_1 \cap \alpha_2
\end{aligned}$$

The intuitive meaning of the state formulas and the until operator $\mathcal{U}$ is as in CTL. In the PDL-like formula $\langle\alpha\rangle\Phi$, the regular I/O-stream expression α specifies a set of finite I/O streams, i.e., finite sequences of concurrent I/O-operations, possibly ending with the symbol $\surd$. Intuitively, $\langle\alpha\rangle\Phi$ holds for a maximal run if it starts with a finite prefix where the data flow matches the conditions specified by α.

Other operators can be derived, e.g., $\Diamond\Phi = \mathit{true}\ \mathcal{U}\ \Phi$ (eventually), $\forall\Box\Phi = \neg\exists\Diamond\neg\Phi$ and $\exists\Box\Phi = \neg\forall\Diamond\neg\Phi$ (always). The dual to the PDL-like modality $\langle\cdot\rangle$ is obtained by $\exists[\alpha]\Phi = \neg\forall\langle\alpha\rangle\neg\Phi$ and $\forall[\alpha]\Phi = \neg\exists\langle\alpha\rangle\neg\Phi$. Intuitively, $[\alpha]\Phi$ holds for a maximal run if all its finite prefixes θ, where the induced I/O-stream belongs to the language given by α, end in a state where Φ holds. The next step operator $\bigcirc$ of LTL/CTL-like logics arises as a special instance of $\langle\cdot\rangle$ by $\bigcirc\Phi = \langle\mathit{true}\rangle\Phi$.

The semantics of a regular data expression α is provided by means of a language $\mathfrak{L}_{\mathcal{N}}(\alpha) \subseteq 2^{IOS}$ where *IOS* denotes the set of all finite I/O-streams, i.e., finite sequences of concurrent I/O-operations, possibly ending with the special symbol $\surd$ denoting that there is no further data flow. We define $\mathfrak{L}_{\mathcal{N}}(g)$ to be the set of all concurrent I/O-operations (N, δ), viewed as words (I/O-streams) of length 1, such that $(N, \delta) \in [\![g]\!]_{\mathcal{N}}$. The language $\mathfrak{L}_{\mathcal{N}}(\mathit{stop})$ is the singleton set $\{\surd\}$. The operators $\cup$, $\cap$ and $\neg$ in the grammar for regular I/O-stream expressions have the standard meaning, i.e., $\cap$ stands for intersection, $\cup$ for union, and $\neg$ for complementation. (Complementation and intersection could be dropped in the syntax of regular I/O-streams expressions without decreasing the expressivity of the logic. We included them in our syntax since there are no closed regular expressions for $\neg\alpha$ or $\alpha_1 \cap \alpha_2$.) The meaning of ; and $*$ agrees with standard concatenation and Kleene closure, except for a special treatment of $\surd$. If $\mathfrak{L}_1, \mathfrak{L}_2 \subseteq 2^{IOS}$ then $\mathfrak{L}_1 ; \mathfrak{L}_2$ arises by the pointwise concatenation $\sigma_1 ; \sigma_2$ of the elements in $\sigma_1 \in \mathfrak{L}_1$ and the elements $\sigma_2 \in \mathfrak{L}_2$ where $\sigma_1 ; \sigma_2 = \sigma_1$ if σ_1 ends with $\surd$. The Kleene closure is then defined in the standard way by $\mathfrak{L}^* = \bigcup \mathfrak{L}^n$ where $\mathfrak{L}^0 = \{\varepsilon\}$ (the language consisting of the empty I/O-stream), $\mathfrak{L}^1 = \mathfrak{L}$ and $\mathfrak{L}^{n+1} = \mathfrak{L} ; \mathfrak{L}^n$.

BTSL formulas over the signature $(AP, \mathcal{N})$ are interpreted over a constraint automaton with the node-set $\mathcal{N}$ and the set AP of atomic propositions. For $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0, AP, L)$, the satisfaction relation $\models_{\mathcal{A}}$ for *BTSL* state formulas is defined in the standard way:

$$\begin{array}{lcl}
q \models_{\mathcal{A}} \mathit{true} & & \\
q \models_{\mathcal{A}} \mathit{ap} & \Longleftrightarrow & \mathit{ap} \in L(q) \\
q \models_{\mathcal{A}} \neg\Phi & \Longleftrightarrow & q \not\models_{\mathcal{A}} \Phi \\
q \models_{\mathcal{A}} \Phi_1 \wedge \Phi_2 & \Longleftrightarrow & q \models_{\mathcal{A}} \Phi_1 \text{ and } q \models_{\mathcal{A}} \Phi_2 \\
q \models_{\mathcal{A}} \exists\varphi & \Longleftrightarrow & \text{there exists a run } \theta \in \mathit{MaxRuns}(q) \text{ s.t. } \theta \models_{\mathcal{A}} \varphi \\
q \models_{\mathcal{A}} \forall\varphi & \Longleftrightarrow & \text{for all runs } \theta \in \mathit{MaxRuns}(q)\text{: } \theta \models_{\mathcal{A}} \varphi
\end{array}$$

The meaning of the path formulas is as follows. If θ is a maximal run then $\theta \models_{\mathcal{A}} \langle\alpha\rangle\Phi$ iff there exists a finite prefix θ' of θ such that $p \models_{\mathcal{A}} \Phi$ for the last state p of θ' and the I/O-stream of θ' belongs to $\mathfrak{L}_{\mathcal{N}}(\alpha)$. The semantics of the until operator is as in CTL. If θ is a maximal run in $\mathcal{A}$ then the satisfaction relation $\theta \models_{\mathcal{A}} (\cdot)$ for *BTSL* run formulas is defined as follows.

If $\theta = q_0 \xrightarrow{N_1,\delta_1} q_1 \xrightarrow{N_2,\delta_2} q_2 \xrightarrow{N_3,\delta_3} \ldots$ is infinite then

$$\theta \models_{\mathcal{A}} \langle\alpha\rangle\Phi \iff \exists j \geq 0 \text{ s.t. } q_j \models_{\mathcal{A}} \Phi \text{ and } (N_1,\delta_1)\ldots(N_j,\delta_j) \in \mathfrak{L}_{\mathcal{N}}(\alpha)$$

If $\theta = q_0 \xrightarrow{N_1,\delta_1} \ldots \xrightarrow{N_k,\delta_k} q_k \xrightarrow{\surd} q_k$ is finite then

$$\begin{array}{rl}
\theta \models_{\mathcal{A}} \langle\alpha\rangle\Phi \iff & \text{either } \exists 0 \leq j \leq k \text{ s.t. } q_j \models_{\mathcal{A}} \Phi \text{ and } (N_1,\delta_1)\ldots(N_j,\delta_j) \in \mathfrak{L}_{\mathcal{N}}(\alpha) \\
& \text{or } q_k \models_{\mathcal{A}} \Phi \text{ and } (N_1,\delta_1)\ldots(N_k,\delta_k)\surd \in \mathfrak{L}_{\mathcal{N}}(\alpha)
\end{array}$$

For θ to be an infinite or finite maximal run with the state sequence $q_0\,q_1\,q_2\ldots$:

$$\theta \models_{\mathcal{A}} \Phi_1 \,\mathcal{U}\, \Phi_2 \iff \exists 0 \leq j < |\theta| \text{ s.t. } q_j \models_{\mathcal{A}} \Phi_2 \wedge \forall 0 \leq i < j.\ q_i \models_{\mathcal{A}} \Phi_2$$

■

Let $\mathit{Sat}_{\mathcal{A}}(\Phi) = \{q \in Q \mid q \models_{\mathcal{A}} \Phi\}$. If $\mathcal{A}$ is clear from the context then we skip the subscript $\mathcal{A}$ and simply write $\models$ and $\mathit{Sat}(\cdot)$. Automaton $\mathcal{A}$ fulfills Φ, denoted $\mathcal{A} \models \Phi$, if $q_0 \models_{\mathcal{A}} \Phi$ for all initial states $q_0 \in Q_0$.

Example 3.1 For a synchronous channel with source node A and sink node B the *BTSL* formula $\forall\Box\forall\langle \mathit{stop} \cup (d_A = d_B)\rangle \mathit{true}$ holds, asserting that all runs in the automaton consist of concurrent I/O-operations where data items are transmitted synchronously from A to B, and possibly end if the components connected to A or B do not provide the corresponding write or read operation. For the FIFO1 channel with source node A and sink node B, the formulas $\forall[\mathit{true}^*; A]\mathit{full}$ and $\forall[\mathit{true}^*; B]\mathit{empty}$ hold, stating that after A's write operation the buffer is full, while after B's read operation the buffer is empty. Also the formula $\forall\Box\neg\exists\langle A \wedge B\rangle \mathit{true}$ holds for the FIFO1 channel stating the impossibility of simultaneous data flow at A and B.

For (the constraint automata of) the network on the left of Fig. 2, the *BTSL* formulas $\forall\Box\neg\exists\langle A \wedge B\rangle \mathit{true}$, $\forall[\mathit{true}^*; A]\forall\langle B\rangle \mathit{true}$, $\forall[\mathit{true}^*; B]\forall\langle A\rangle \mathit{true}$ and $\forall\langle \mathit{true}^*\rangle\forall\langle d_A = d \cup d_B =$

$d\rangle$ *true* hold. (The d in the picture denotes that the upper buffer is filled with the data item d in the initial configuration.) The former three formulas state that data flow at A and B alternates, while the latter formula asserts that only data item d, observed at A or B, is possible.

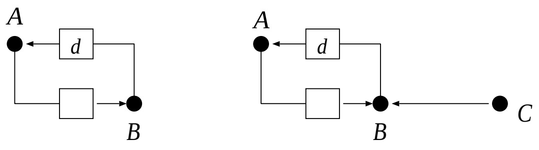


Fig. 2. Two Reo networks

While the network on the left has no terminal states, and thus, data flow is always infinite, the source node C in the network on the right may write into the upper buffer which yields the configuration where both buffers are filled and data flow stops. Hence, the network on the right fulfills the formulas $\forall[\mathit{true}^*\,;A]\forall\langle(B\,;A)\cup(C\,;\mathit{stop})\rangle\mathit{true}$, $\forall[\mathit{true}^*\,;B]\forall\langle A\rangle\mathit{true}$ and $\forall[\mathit{true}^*\,;C]\mathit{both\_buffers\_full}$, where *both_buffers_full* is an atomic proposition with the obvious semantics.

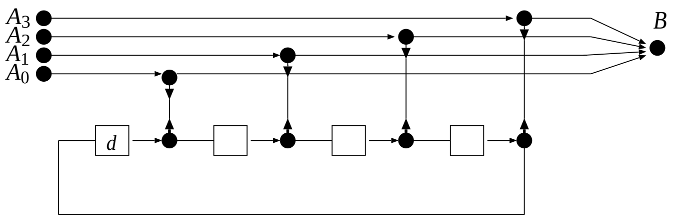


Fig. 3. A sequencer

Fig. 3 shows the network for a sequencer, built out of 4 FIFO1 channels and several synchronous channels and drains that allows the A_i's to send messages to B in the order $A_0A_1A_2A_3A_0A_1A_2A_3\ldots$. This property can be formalized by the formulas $\neg\exists\langle(\mathit{true}^*\,;A_i\,;A_j)\rangle\mathit{true}$ where $0 \le i < j$ and $j \neq i+1$ (modulo 3). Other properties that hold for the sequencer are $\forall[\mathit{true}^*\,;(\neg\mathit{stop}\cap\neg B)]\mathit{false}$, $\forall[\mathit{true}^*\,;A_i]\mathit{filled(i+1)}$ and $\forall\Box\big(\mathit{filled(i+1)}\rightarrow\exists\langle d_{A_i}=d_B\rangle\mathit{true}\big)$ where *filled(i+1)* is an atomic proposition stating that the $(i+1)$-st buffer is filled (modulo 3).

The terminal states of a constraint automaton are characterized by the formula $\Phi_{terminal} = \exists\langle\mathit{stop}\rangle\mathit{true}$. ■

4 Symbolic *BTSL* Model Checking

The *BTSL* model checking problem takes as input a Reo network, possibly together with constraint automata that specify the interfaces of the components that are connected to the source and sink nodes of the network, and a *BTSL* formula which has to be checked for the network. The automata for the components that are connected to the sink or source nodes of the network describe the environment in which the network operates. They may restrict the nondeterminism in the automaton for the network, since certain transition instances (concurrent I/O-operations) might become impossible due to the behavioral interfaces of the components. After connecting a sink and source node A of the network with a port of a component, A is treated as a mixed node. Thus, the automata for the component might also decrease the set of terminal states. In case nothing is known about the potential behaviors of the components that will be coordinated by the network, these automata can be skipped, in which case all possible interactions of the sink and source nodes will be taken into account for the analysis.

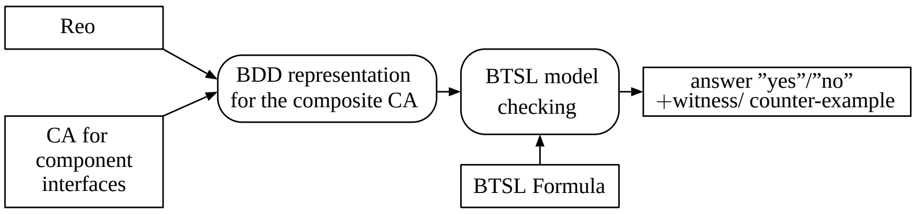

Reo
CA for component interfaces
BDD representation for the composite CA
BTSL model checking
BTSL Formula
answer "yes"/"no" +witness/ counter-example

Fig. 4. Model Checking schema

The schema of our model checker is depicted in Fig. 4. The first step is to generate an appropriate representation of the constraint automaton associated with the network, possibly within the environment given by the automata for the components. The goal of the second step is to verify or falsify whether for the generated constraint automata a given *BTSL* formula holds in all initial states. For certain formula types the model checker can return a witness (e.g., a run θ with $\theta \models \varphi$ if the formula to be checked is $\exists\varphi$) or a counterexample (e.g., a run θ with $\theta \not\models \varphi$ if the formula to be checked is $\forall\varphi$).

In the remainder of this section, we report on a symbolic *BTSL* model checker. We first summarize the main steps of the *BTSL* model checking algorithm and then explain its symbolic realization.

The model checking algorithm. *BTSL* model checking relies on a combination of the CTL model checking algorithm [11] with automata-based approaches. Given a constraint automata $\mathcal{A}$ and *BTSL* state formula Φ, the idea is an iterative computation of the satisfaction sets $Sat_{\mathcal{A}}(\Psi)$ for the sub-state-formulas Ψ of Φ.

The treatment of the propositional logic fragment is obvious. The satisfaction sets for formulas $\exists(\Phi_1\ \mathcal{U}\ \Phi_2)$ or $\forall(\Phi_1\ \mathcal{U}\ \Phi_2)$ are obtained as in CTL, only slight modifications are necessary for a correct treatment of terminal states. For formulas of the form $\exists\langle\alpha\rangle\Psi$ or $\exists[\alpha]\Psi$, [2] we first apply standard algorithms to generate a nondeterministic finite automata (NFA) $\mathcal{Z}$ for the regular I/O-stream expression α. The alphabet of $\mathcal{Z}$, i.e., the range of the transition labels in $\mathcal{A}$, is $IOC \cup \{\surd\}$. In fact, beside the special $\surd$-transitions, $\mathcal{Z}$ can be viewed as a constraint automata $\mathcal{Z} = (Z, \mathcal{N}, \longrightarrow, Z_0, Z_F)$ with an additional set Z_F of final (accept) states. The atomic propositions and labeling function are irrelevant for $\mathcal{Z}$. By the special role of the end symbol $\surd$, we may assume that $\mathcal{Z}$'s state space Z contains a subset $Z_{\surd}$ such that (i) $z \xrightarrow{\surd} z'$ implies $z' \in Z_{\surd}$, (ii) $z \xrightarrow{g} z' \in Z_{\surd}$ implies $g = \surd$, (iii) the states in $Z_{\surd}$ do not have successors.

Given $\mathcal{A}$ and $\mathcal{Z}$, we then built the product $\mathcal{A} \times \mathcal{Z}$ where the states are pairs (q,z) consisting of a state q in $\mathcal{A}$ and a state z in $\mathcal{Z}$. The transitions in $\mathcal{A} \times \mathcal{Z}$ are obtained by the

[2] We explain here an algorithm for $\exists[\alpha]\Psi$. The treatment of formulas $\forall\langle\alpha\rangle\Psi$ is obtained by the duality law $\forall\langle\alpha\rangle\Psi \equiv \neg\exists[\alpha]\neg\Psi$.

following rules:

$$\frac{q \xrightarrow{g_1}_{\mathcal{A}} q' \ \wedge \ z \xrightarrow{g_2}_{\mathcal{Z}} z' \ \wedge \ g_1, g_2 \in IOC}{(q,z) \xrightarrow{g_1 \wedge g_2}_{\mathcal{A}\times\mathcal{Z}} (q',z')} \qquad \frac{q \text{ is terminal in } \mathcal{A} \ \wedge \ z \xrightarrow{\surd}_{\mathcal{Z}} z'}{(q,z) \xrightarrow{\surd}_{\mathcal{A}\times\mathcal{Z}} (q,z')}$$

where we use the subscripts $\mathcal{A}$, $\mathcal{Z}$ or $\mathcal{A}\times\mathcal{Z}$ for the transition relations in $\mathcal{A}$, $\mathcal{Z}$ and $\mathcal{A}\times\mathcal{Z}$, respectively. The product $\mathcal{A}\times\mathcal{Z}$ is equipped with two atomic propositions $sat(\Psi)$ and *final* and the labeling function that assigns $sat(\Psi)$ to all states (q,z) where $q \models_{\mathcal{A}} \Psi$ and *final* to all states (q,z) where $z \in Z_F$. The following proposition (see appendix for the proof) provides a reduction to CTL.

Proposition 4.1 (Reduction to CTL)

(a) $q \models_{\mathcal{A}} \exists\langle\alpha\rangle\Psi$ *iff there exists* $z_0 \in Z_0$ *with* $(q,z_0) \models_{\mathcal{A}\times\mathcal{Z}} \exists\Diamond(sat(\Psi)\wedge final)$

(b) If $\mathcal{A}$ *is deterministic then* $q \models_{\mathcal{A}} \exists[\alpha]\Psi$ *iff* $(q,z_0) \models_{\mathcal{A}\times\mathcal{Z}} \exists\Box(sat(\Psi)\vee\neg final)$ *where* z_0 *is the initial state of* $\mathcal{Z}$.

Part (a) of Prop. 4.1 allows to compute $Sat(\exists\langle\alpha\rangle\Psi)$ by means of a backward reachability analysis in $\mathcal{A}\times\mathcal{Z}$. For $Sat(\exists[\alpha]\Psi)$, the second part of Prop. 4.1 suggests to switch from $\mathcal{Z}$ to an equivalent deterministic finite automata (DFA) and to search for cycles in a subgraph of the product of $\mathcal{A}$ and the DFA. However, the determinization of $\mathcal{Z}$ (which can cause an exponential blow-up) can be avoided by applying Algorithm 1.

Algorithm 1 Computation of $Sat(\exists[\alpha]\Psi)$

```
construct an NFA $\mathcal{Z}$ for $\alpha$ and built the product $\mathcal{A}\times\mathcal{Z}$;
$V := \{(q,z) \in Q\times Z \mid q \in Sat(\Psi) \vee z \notin Z_F\}$;
repeat
  $V' := V$;
  $R := \{(q,z) \mid \forall$ transition instances $q \xrightarrow{N,\delta}_{\mathcal{A}} q'\ \exists z \xrightarrow{N,\delta}_{\mathcal{Z}} z'$ s.t. $(q',z') \notin V\}$;
  $V := (V \setminus R) \cup \{(q,z) \in V \mid q$ terminal $\wedge z \in Z_{\surd}\}$
until $(V' = V)$;
return $\{q \in Q \mid (q,z_0) \in V$ for all $z_0 \in Z_0\}$
```

Proposition 4.2 *Algorithm 1 computes the set of states* $q \in Q$ *where* $q \models_{\mathcal{A}} \exists[\alpha]\Psi$.

Proof. Let $\overline{V}$ be the set of states (q,z) that belong to V when the repeat-loop terminates. Furthermore, let $V_0 = \{(q,z) \mid q \in Sat(\Psi) \vee z \notin Z_F\}$, $W_0 = Q\times Z\setminus V_0$ and let W_i be the set of states that are removed from V in the i-th iteration. Then,

$$\overline{V} = \textstyle\bigcap_{i\geq 0} V_i = V_n$$

where $V_{i+1} = V_i \setminus W_{i+1}$ and n is the number of iterations. Moreover, we have:

(i) for all $(q,z) \in \overline{V}$ where q is non-terminal there exists a transition instance $q \xrightarrow{N,\delta} q'$ such that $(q',z') \in \overline{V}$ for all $z \xrightarrow{N,\delta} z'$.

(ii) for all $(q,z) \in W_i$ and for all runs $q = q_0 \xrightarrow{N_1,\delta_1} \ldots \xrightarrow{N_k,\delta_k} q_m$ of length $m \geq i$ in $\mathcal{A}$ there exists a run $z = z_0 \xrightarrow{N_1,\delta_1} \ldots \xrightarrow{N_k,\delta_k} z_k$ of length $k \leq \min\{i,m\}$ such that $(q_k,z_k) \notin V_0$, i.e., $q_k \not\models_{\mathcal{A}} \Psi$ and $z_k \in Z_F$.

Let us now assume that q_0 is a state contained in the set returned by Algorithm 1. Then, $(q_0,z_0) \in \overline{V}$ for all initial states z_0 in $\mathcal{Z}$. We successively apply (i) to obtain a maximal run in $\mathcal{A}$

$$\theta = q_0 \xrightarrow{N_1,\delta_1} q_1 \xrightarrow{N_2,\delta_2} \dots$$

such that for all runs

$$z_0 \xrightarrow{N_1,\delta_1} z_1 \xrightarrow{N_2,\delta_2} \dots$$

in $\mathcal{Z}$ for the same I/O-stream, we have $(q_i,z_i) \in \overline{V}$ for all indices i. Since $\overline{V} \subseteq V_0$ we obtain $\theta \models_{\mathcal{A}} [\alpha]\Psi$, and thus, $q_0 \models_{\mathcal{A}} \exists[\alpha]\Psi$.

We now consider a state $q_0 \in Q$ such that $q_0 \models_{\mathcal{A}} \exists[\alpha]\Psi$. Let

$$\theta = q_0 \xrightarrow{N_1,\delta_1} q_1 \xrightarrow{N_2,\delta_2} \dots$$

be a maximal run in $\mathcal{A}$ such that $\theta \models_{\mathcal{A}} [\alpha]\Psi$. W.l.o.g. θ has minimal length under all runs $\theta' \in \mathit{MaxRuns}(q_0)$ where $\theta' \models_{\mathcal{A}} \exists[\alpha]\Psi$. If we assume that $(q_0,z_0) \notin \overline{V}$ for some $z_0 \in Z_0$, say $(q_0,z_0) \in W_i$, then $|\theta| \geq i$ and by (ii) there exists $k \leq i$ and a run

$$z_0 \xrightarrow{N_1,\delta_1} \dots \xrightarrow{N_k,\delta_k} z_k$$

in $\mathcal{Z}$ such that $(q_k,z_k) \notin V_0$. Hence, $z_k \in Z_F$ and $q_k \not\models_{\mathcal{A}} \Psi$. But then $(N_1,\delta_1)\dots(N_k,\delta_k) \in \mathfrak{L}_{\mathcal{N}}(\alpha)$ and $\theta \not\models_{\mathcal{A}} [\alpha]\Psi$. Contradiction. This yields $(q_0,z_0) \in \overline{V}$ for all $z_0 \in Z_0$. Hence, q_0 is in the set of states returned by Algorithm 1. □

The complexity of the algorithms to compute the satisfaction sets of $\exists\langle\alpha\rangle\Psi$ and $\forall[\alpha]\Psi$ are polynomial in the size of $\mathcal{A}$ and $\mathcal{Z}$. Thus, the overall time complexity of *BTSL* model checking is polynomial in the size of $\mathcal{A}$ and the length of the input formula Φ, provided the regular I/O-stream expressions in Φ are ordinary regular expressions, i.e., do not use the complementation or intersection operator, since they can cause an exponential blow-up in the construction of $\mathcal{Z}$ from α.

Symbolic implementation. We now summarize the main features of our symbolic *BTSL* model checker with binary decision diagrams (BDDs), see e.g. [8,17,16,19]. BDDs are a data structure for switching functions $f : \mathit{Eval}(x_1,\dots,x_n) \to \{0,1\}$ where $x_1,\dots,x_n$ are boolean variables and $\mathit{Eval}(x_1,\dots,x_n)$ denotes the set of evaluations for $x_1,\dots,x_n$. To represent a constraint automaton $\mathcal{A} = (Q,\mathcal{N},\longrightarrow,Q_0,AP,L)$ by a BDD, we fix a binary encoding of the states, i.e., we embed Q into $\{0,1\}^n$ by an injective function $\mathit{bin} : Q \to \{0,1\}^n$ where $n = \lceil \log|Q| \rceil$, choose boolean state variables $q_1,\dots,q_n$ and then identify each state q with the evaluation for $q_1,\dots,q_n$ given by $\mathit{bin}(q)$. In the same way, we may encode the data items by bit tuples. For simplicity, we assume here the boolean data domain $\mathit{Data} = \{0,1\}$ and treat the symbols d_A and the nodes $A \in \mathcal{N}$ as boolean variables.

In the sequel, let $\mathcal{N} = \{A_1,\dots,A_k\}$ and $d_i = d_{A_i}$, $i = 1,\dots,k$. We write $\overline{A}$ and $\overline{d}$ for the variable tuples $(A_1,\dots,A_k)$ and $(d_1,\dots,d_k)$, respectively. The transition relation $\longrightarrow$ can be identified with its characteristic function and viewed as a switching function $T_{\mathcal{A}} : \mathit{Eval}(\overline{q},\overline{A},\overline{d},\overline{q}') \to \{0,1\}$, where the variable tuple $\overline{q} = (q_1,\dots,q_n)$ encodes the starting state, $\overline{q}' = (q_1',\dots,q_n')$ the target state, while $\overline{A}$ and $\overline{d}$ serve to represent the concurrent I/O-operations. For instance, the transition relations of the constraint automata for a synchronous channel with source node A and sink node B and a synchronous drain are given by:

$$T_{\text{sync\_channel}}(q_1, A, B, d_A, d_B, q_1') = q_1 \wedge A \wedge B \wedge (d_A \leftrightarrow d_B) \wedge q_1'$$
$$T_{\text{sync\_drain}}(q_1, A, B, d_A, d_B, q_1') = q_1 \wedge A \wedge B \wedge q_1'$$

For a FIFO1 channel we have to encode three states, say $bin(q) = 00$, $bin(q(1)) = 11$ and $bin(q(0)) = 10$, and then may represent the automaton by

$$(\neg q_1 \wedge \neg q_2 \wedge A \wedge \neg B \wedge (q_2' \leftrightarrow d_A) \wedge q_1') \;\vee\; (q_1 \wedge \neg A \wedge B \wedge (q_2 \leftrightarrow d_B) \wedge \neg q_1' \wedge \neg q_2')$$

The BDD-representation for the transition relation of a Reo network can be constructed in a compositional manner, by mimicking Reo's composition operators with corresponding operators on constraint automata and applying the analogous symbolic operations for manipulating switching functions. We will briefly consider the join operator which allows to collapse two nodes into a single node. Using some appropriate renaming of nodes, Reo's join operator can be reduced on the automata level to a product construction that "synchronizes" the data flow at the common nodes of the given constraint automata (see [6]). If $\mathcal{A}_1$ and $\mathcal{A}_2$ are constraint automata with node sets $\mathcal{N}_1$ and $\mathcal{N}_2$, respectively, then the concurrent I/O-operations in the product $\mathcal{A}_1 \times \mathcal{A}_2$ are given by the transition instances obtained by the following synchronization rule and two interleaving rules:

$$\frac{q_1 \xrightarrow{g_1}_{\mathcal{A}_1} p_1, \quad q_2 \xrightarrow{g_2}_{\mathcal{A}_2} p_2}{(q_1, q_2) \xrightarrow{g_1 \wedge g_2}_{\mathcal{A}_1 \times \mathcal{A}_2} (p_1, p_2)}$$

$$\frac{q_1 \xrightarrow{g_1}_{\mathcal{A}_1} p_1}{(q_1, q_2) \xrightarrow{g_1 \wedge \neg \mathcal{N}_2}_{\mathcal{A}_1 \times \mathcal{A}_2} (p_1, q_2)} \qquad \frac{q_2 \xrightarrow{g_2}_{\mathcal{A}_2} p_2}{(q_1, q_2) \xrightarrow{g_2 \wedge \neg \mathcal{N}_1}_{\mathcal{A}_1 \times \mathcal{A}_2} (q_1, p_2)}$$

where $\neg \mathcal{N}_i$ stands short for $\bigwedge_{A \in \mathcal{N}_i} \neg A$.

These rules can be realized in a symbolic way by putting $T_{\mathcal{A}_1 \times \mathcal{A}_2} = (T_{\mathcal{A}_1} \wedge T_{\mathcal{A}_2}) \vee (T_{\mathcal{A}_1} \wedge \neg \mathcal{N}_2 \wedge id_{\mathcal{A}_2}) \vee (T_{\mathcal{A}_2} \wedge \neg \mathcal{N}_1 \wedge id_{\mathcal{A}_1})$, where $id_{\mathcal{A}} = \bigwedge_{q \in Q}(q \leftrightarrow q')$ and Q is in the state space of $\mathcal{A}$.

Beside the transition relation, we also need a BDD-represent of the labeling function. This can be done by representing the characteristic function of $Sat(ap) = \{q \in Q \mid ap \in L(q)\}$ by a BDD for the induced function $f_{ap} : Eval(\overline{q}) \rightarrow \{0, 1\}$. BDD-representations f_Ψ for the satisfaction sets $Sat(\Psi)$ of the subformulas Ψ of Φ are obtained by reformulating the *BTSL* model checking algorithm in a symbolic way with boolean operators and applying the corresponding BDD synthesis algorithms. A symbolic reformulation of Algorithm 1 is shown in Algorithm 2 where it is assumed that the BDD f_Ψ for $Sat(\Psi)$ and a BDD-representation *Terminal* for the set of terminal states has already been constructed. We use the variable tuple $\overline{q} = (q_1, \ldots, q_n)$ to encode the states in $\mathcal{A}$ and $\overline{z} = (z_1, \ldots, z_m)$ for the states in $\mathcal{Z}$. Subsets V of $Q \times Z$ are encoded by the variables in $\overline{q}$ and $\overline{z}$. The notation $V(\overline{q}', \overline{z}')$ means that the variables of V are renamed into their primed copies. The sets Z_0, Z_F and $Z_{\surd}$ are represented by BDDs with the variables $\overline{z}$.

5 Examples and results

We applied the *BTSL* model checker to a couple of examples. We will report here on two case studies. All results were achieved on a Pentium IV, 1.8GHz, 1.5GB RAM with

Algorithm 2 Computation of the symbolic representation $f_{\exists[\alpha]\Psi}$ for $Sat(\exists[\alpha]\Psi)$

```
construct an NFA $\mathcal{Z}$ for $\mathcal{A}$ and generate BDD-representations $T_{\mathcal{Z}}$ for the transition relation of $\mathcal{Z}$ and for the sets $Z_0$, $Z_F$ and $Z_{\surd}$;
$V := f_\Psi \vee \neg Z_F$;
repeat
    $V' := V$;
    $R := \forall\overline{q}\,\forall\overline{A}\,\forall\overline{d}.\big(\,T_{\mathcal{A}} \Rightarrow \exists\overline{z}'.(T_{\mathcal{Z}} \wedge V(\overline{q}',\overline{z}'))\big)$;
    $V := (V \wedge \neg R) \vee (V \wedge Terminal \wedge Z_{\surd})$
until $(V' = V)$;
return $\forall\overline{z}.(Z_0 \Rightarrow V)$          (* symbolic representation $f_{\exists[\alpha]\Psi}$ for $Sat(\exists[\alpha]\Psi)$ *)
```

Mandriva Linux and kernel 2.6.12. The tool was written in C++, compiled with GCC4.0.3 and uses JINC [18] as library for binary decision diagrams.

Example 5.1 [Dining philosophers] The first example describes the well-known dining philosophers scenario, modelled in Reo as in [1], see Fig. 5.

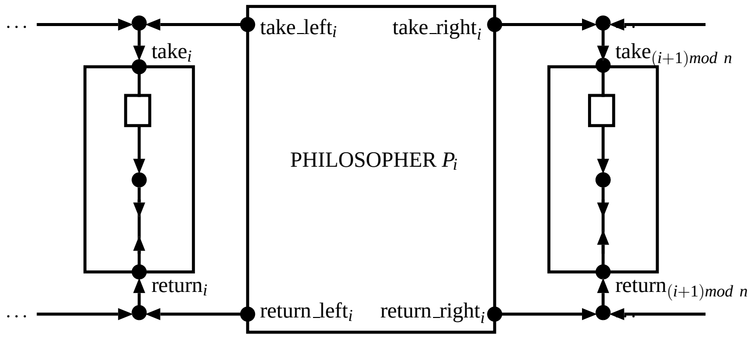


Fig. 5. Dining philosophers scenario

The interface of philosopher i has four output ports *take_left*$_i$, *take_right*$_i$, *return_left*$_i$ and *return_right*$_i$ that serve to take and return the chopsticks on the left and right of the philosopher. The chopsticks are modelled by a FIFO1 channel and synchronous drain. The constraint automata for the interfaces of the philosophers and the chopsticks are shown in Fig. 6.

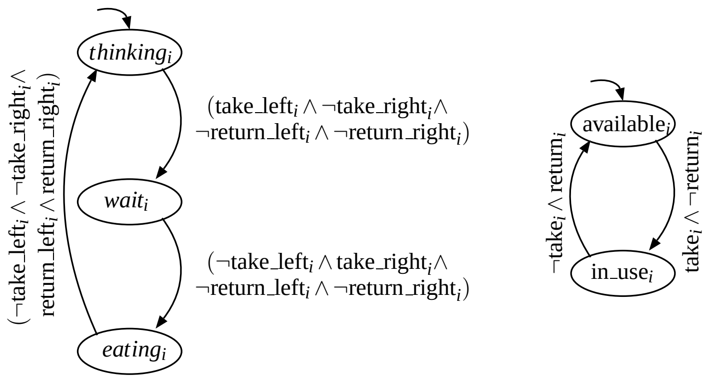

$thinking_i$
$wait_i$
$eating_i$
$(\text{take\_left}_i \wedge \neg\text{take\_right}_i \wedge \neg\text{return\_left}_i \wedge \neg\text{return\_right}_i)$
$(\neg\text{take\_left}_i \wedge \text{take\_right}_i \wedge \neg\text{return\_left}_i \wedge \neg\text{return\_right}_i)$
$(\neg\text{take\_left}_i \wedge \neg\text{take\_right}_i \wedge \text{return\_left}_i \wedge \text{return\_right}_i)$
available$_i$
in_use$_i$
$\neg\text{take}_i \wedge \text{return}_i$
$\text{take}_i \wedge \neg\text{return}_i$

Fig. 6. CA for philosopher and chopstick

Table 1 illustrates the efficiency of the symbolic approach to construct the BDD-representation of the constraint automaton $\mathcal{A}$ for the whole system by the symbolic join-operation. The first column "size" shows the number of philosophers. The second column "time" shows the time needed for the synthesis phase, while the last column "reachable time" refers to the time needed to compute the reachable fragment of $\mathcal{A}$. The other two columns refer to the size of the generated BDD for $\mathcal{A}$ and the maximal size of the BDDs generated during the symbolic computation.

Size	Time	BDD Nodes	Peak	Reach Time
200	0.98s	33146	285523	0.24s
400	2.18s	66546	572523	0.45s
800	4.97s	133346	1146523	0.86s
1600	12.69s	266946	2294523	1.81s
3200	35.12s	534146	4590523	3.96s
6400	112.21s	1068546	9182523	8.53s

Table 1
Synthesis results for the dining philosophers

To give an impression of the size of the state space: the reachable part of the CA for 800 philosophers consists of about 10^{306} states. Several properties have been checked for this model of the dining philosophers. Table 2 shows the results for three *BTSL* formulas. The columns refer to the number of philosophers, number of steps in the model checking procedure namely the number of iteration within the fixpoint computation and the total amount of time needed to verify (or falsify) the given formula.

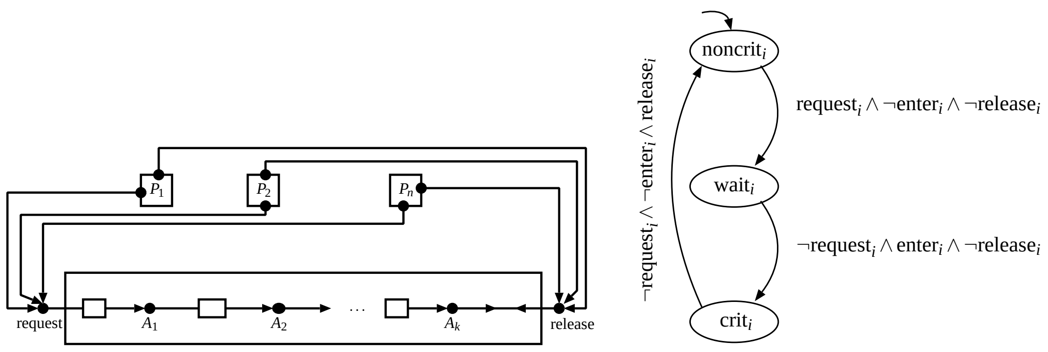
Size	Formula	Steps	Time	Peak
200	$\forall\Box(\neg(eat_{100} \wedge eat_{101}))$	199	17.78s	5169232
200	$\forall\Box[\exists\langle true^*;\text{take\_right}_i\rangle true]$	798	135.04s	34762951
3200	$\exists\langle true^*;\text{take}_i;\text{take}_{(i+1) mod\ n}\rangle eat_i$	5	16.56s	9303687

Table 2
Model Checking results for the dining philosophers

The second formula does not hold since there is the run where all philosophers take their left chopstick and then wait forever for the missing right chopstick. This deadlock situation has been found with 798 iterations by means of a backward analysis. Computing the reachable part first by means of a forward analysis, the deadlock can be found in 403 steps within 13.92s only.

Example 5.2 [Mutual exclusion] The second example is the component connector shown in Fig. 7 that realizes a mutual exclusion protocol for n parallel processes $(P_1, \ldots, P_n)$ where at each time instance at most k may perform their critical actions.

We assume here that the behavioural interface of each component P_i is represented by the constraint automaton also depicted in Fig. 7.



P_1
P_2
P_n
request
A_1
A_2
A_k
release
noncrit_i
wait_i
crit_i
$\text{request}_i \wedge \neg\text{enter}_i \wedge \neg\text{release}_i$
$\neg\text{request}_i \wedge \text{enter}_i \wedge \neg\text{release}_i$
$\neg\text{request}_i \wedge \neg\text{enter}_i \wedge \text{release}_i$

Fig. 7. Mutual exclusion scenario and CA for one process

Table 3 summarizes the results for the generation of the BDD-representation, where n is the number of processes and k the maximum number of processes allowed to be in their critical section at the same time. For 200 processes and $k = 60$ this CA consists of more than $5 \cdot 10^{119}$ reachable configurations.

n	k	Time	BDD Nodes	Peak	Reach Time
200	5	4.34s	9617	1735363	0.15s
200	20	5.74s	11907	2295538	0.89s
200	60	9.38s	17986	3789338	9.64s
400	5	17.17s	18617	5933461	0.29s
400	20	20.14s	20907	7045636	1.64s
400	60	28.64s	26986	10011436	11.77s
800	5	62.99s	36617	20508457	0.58s
800	20	69.26s	38907	22724632	3.07s
800	60	85.99s	44986	28634432	20.58s

Table 3
Synthesis results for the mutual exclusion network

We performed the analysis with several *BTSL* formulas. Table 4 shows the results for three formulas: $\Phi_1 = \forall[\text{request}^*]\ (\bigwedge_{1\leq i\leq n} \neg crit_i)$,

$\Phi_2 = \exists\langle true^*; \text{enter}_1; A_1; (\text{enter}_2 \wedge A_2); A_1; (\text{enter}_3 \wedge A_3)\rangle(crit_1 \wedge crit_2 \wedge crit_3)$ and

$\Phi_3 = \forall[true^*; \text{enter}_1; (\neg\text{release})^*; \ldots; \text{enter}_k; (\neg\text{release})^*)]\neg\exists\langle(\neg\text{release})^*; \text{enter}_{k+1}\rangle true$.

6 Conclusion

The purpose of the paper was to explain the functionality and foundations of our model checker for Reo networks. The efficiency has been illustrated by two examples that show that our model checking approach can handle even very large networks with up to 10^{1200} configurations in a reasonable amount of time. Given the wide range of applications of the Reo framework, see e.g. [13,20,9], we believe that our model checker yields an important

Processes (n)	Semaphors (k)	Time (Φ_1)	Time (Φ_2)	Time (Φ_3)
200	5	0.80s	0.15s	0.68s
200	20	0.86s	0.19s	0.82s
200	60	0.82s	0.38s	1.89s
400	5	1.74s	0.31s	1.47s
400	20	1.82s	0.35s	1.58s
400	60	1.43s	0.53s	2.53s
800	5	4.57s	0.62s	3.69s
800	20	4.58s	0.65s	3.63s
800	60	3.62s	0.87s	4.61s

Table 4
Model Checking results for the mutual exclusion

contribution for formal reasoning about exogeneous coordination models. Beside further optimizations to increase efficiency and case studies, we will extend our implementation to reason about real-time constraints with the logic TDSL [3] or a branching time version thereof and about dynamic reconfigurations by means of the logic considered in [10] or other formal frameworks for Reo's dynamic operations.

References

[1] F. Arbab, Abstract Behavior Types: A Foundation Model for Components and Their Composition, In [7],33-70, 2003.

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

[3] F. Arbab and C. Baier and F. de Boer and J. Rutten, Models and Temporal Logics for Timed Component Connectors, In Proc. SEFM'04, IEEE CS Press, 2004.

[4] F. Arbab and C. Baier and F. de Boer and J. Rutten, Models and Temporal Logics for Timed Component Connectors, Software and Systems Modelling (to appear), 2006.

[5] F. Arbab and J.J.M.M. Rutten, A coinductive calculus of component connectors, In Proc. 16th WADT, volume 2755 of LNCS, pages 35-56, 2003.

[6] 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.

[7] F.S. de Boer and M.M. Bonsangue and S. Graf and W.-P. de Roever, Formal Methods for Components and Objects, LNCS 2852, Springer, 2003.

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

[9] D. Clarke and D. Costa and F. Arbab, Modeling Coordination in Biological Systems, In Proc. of the Int. Symposium on Leveraging Applications of Formal Methods, 2004.

[10] Dave Clarke, Reasoning about Connector Reconfiguration II: Basic reconfiguration Logic, In Proc. FSEN'05, Teheran, Electronic Notes in Theoretical Computer Science, 2005.

[11] E. Clarke and E. Emerson and A. Sistla, Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Transactions on Programming Languages and Systems, 8(2):244-263, April 1986.

[12] E. Clarke and O. Grumberg and D. Peled, Model Checking, MIT Press, 1999.

[13] N. Diakov and F. Arbab, Compositional Construction of Web Services Using Reo, In Proc. International Workshop on Web Services: Modeling, Architecture and Infrastructure (ICEIS 2004), Porto, Portugal, April 13-14, 2004.

[14] E. Emerson and C. Lei, Modalities for Model Checking: Branching Time Strikes Back (extended abstract), In Proc. 12th Annual ACM Symposium on Principles of Programming Languages (POPL), pages 84-96, SIGPLAN, ACM Press, 1985.

[15] M. Fischer and J. Ladner, Propositional dynamic logic of regular programs, Journal of Computer and Systems Sciences, 18:194-211, 1979.

[16] G. Hachtel and F. Somenzi, Logic Synthesis and Verification Algorithms, Kluwer Academic Publishers, 1996.

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

[18] J. Ossowski, JINC, a bdd library (to be published), www.jossowski.de.

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

[20] Z. Zlatev and N. Diakov and S. Pokraev, Construction of Negotiation Protocols for E-Commerce Applications, ACM SIGecom Exchanges, 5:2):11-22, November 2004.