Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 255 (2009) 137–158
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Automated Analysis of Reo Circuits using Symbolic Execution

Bahman Pourvatan [1]

Computer Engineering Dept., Amirkabir University of Technology, Tehran, Iran
LIACS, Leiden University, Netherlands

Marjan Sirjani[2]

School of Computer Science, Reykjavik University, Reykjavik, Iceland
ECE Dept., University of Tehran, Tehran, Iran

Hossein Hojjat[3]

EPFL, Lausanne, Switzerland

Farhad Arbab[4]

LIACS, Leiden University,Netherlands,
CWI, Netherlands

Abstract

Reo is a coordination language that can be used to model different systems. We propose a technique for symbolic execution of Reo circuits using Constraint Automata and more specifically exploiting their data constraints. This technique enables us to obtain the relations among the data passing through different nodes in a circuit and also infer coordination patterns. As an alternative to constructing the symbolic execution tree, we propose an algorithm similar to the algorithms used for converting deterministic finite automata to regular expressions. Our technique can be used for analyzing data-dominated systems whereas the model checking approach is best suited for the study of control-dominated applications. Deadlocks, which may involve data values, can also be checked. We illustrate the technique on a set of data dominated circuits as well as a non-trivial critical section problem. A tool is implemented to automate the proposed technique.

Keywords: Symbolic Execution, Reo, Constraint Automata, Coordination Languages, Program Verification.

[1] Email: pourvatan@aut.ac.ir
[2] Email:marjan@ru.is
[3] Email:hossein.hojjat@epfl.ch
[4] Email:farhad@cwi.nl

doi:10.1016/j.entcs.2009.10.029

1 Introduction

Reo [1] is an exogenous coordination language. In Reo, complex connectors are compositionally built out of simpler ones. The simplest connectors in Reo are a set of channels with well-defined behavior. Reo connectors are visually represented as circuits similar to electronic circuits and show how components are inter-connected. The emphasis in Reo is on the connectors, and the synchronization and communication among components, and not on the internal behavior of components. Constraint Automata [3,7] are introduced as compositional semantics for Reo. Using Constraint Automata (CA) we can analyze the behavior of Reo circuits.

Reo is introduced as a coordination language and it can be used in various kinds of applications. Reo and Constraint Automata are used as an ADL (Architectural Description Language) in [4], they can be used in modeling hardware and system level designs [20,21], and in modeling web services [24,25]. In these applications, Reo is generally used to show the communication and synchronization, and Constraint Automata are used to model the components. In this way, the behavior of the whole system can be compositionally constructed using the Constraint Automata of its constituents.

For analysis of Reo circuits, a model checker is presented in [13]. Our symbolic execution technique and tool constitutes a simpler yet powerful and more efficient analyzer which can be used as an alternative or extension for this model checker. When applicable, it can be used to derive the symbolic output of a circuit in terms of its input data, which can also reveal possible deadlocks/livelocks at less computational cost than model checking. Whereas in model checking you can express more complicated properties and pay the higher computational cost for its analysis, symbolic execution of Reo circuits can be effectively used for instance when Reo is used for modeling hardware and system design. Similar techniques can be used for slicing reductions and test case generation for Reo and CA.

The main idea behind symbolic execution [12] is to use symbolic values, instead of actual data, as input values and to represent the values of program variables as symbolic expressions. Consequently, the output values computed by a program are expressed as a function of the symbolic input values [11]. While symbolic execution tools are comparable with model checkers, they are mostly used in data-dominated rather than control-dominated applications. The complexity of data-dominated applications does not necessarily arise from complex data structures and data types. Rather, their complexity arises from nontrivial interdependencies among data items in the applications.

For symbolic execution of Reo circuits we use Constraint Automata, which allows us to use known algorithms and techniques in the domain of automata theory. For Constraint Automata, we consider an *execution path*, which is a path from the entry to the exit of a program, as a path (or run) from an initial state back to an initial state of the CA. Speaking in the context of Reo, this means that all enabled nodes are fired and the circuit returns to its initial configuration, i.e., a transaction, an interaction pattern, or a coordination task is completed. For simplicity, and without loss of generality, we consider deterministic Constraint Automata, with a single initial state. We generate the regular expression of a CA, assuming the initial state as the final state [9]. For that, we define the set of regular expressions associated with CA considering how synchrony and asynchrony are both modeled in CA, and showing how data constraints can be represented in the regular expressions. Our approach relies on the manipulation of the data constraints in different execution paths.

Two main difficulties in performing symbolic execution are (1) handling complicated data structures and data types; and (2) dealing with a potentially infinite number of symbolic execution paths. Initially, we abstract from complicated data structures and data types as they are not regularly considered in Reo and Constraint Automata and we focus on deriving nontrivial interdependencies among data. Instead of dealing with an infinite number of execution paths, here we traverse each cycle a certain number of times which depends on the length of the longest path between memory cells in their transitive relation (the memory cells in each cycle are in a relation by appearing in the same data constraint), yielding a finite number of paths. This is enough to give us the relations among the elements of the input and the output data streams passing through the circuit (represented symbolically).

Contribution of the work. We provide an automated analysis technique and its supporting tool for Reo circuits based on the symbolic execution approach. Reo is used for different applications but only a few tools are provided for its analysis [15,13]. Although our technique does not support verifying temporal logic properties, it provides reachability analysis and can reveal possible deadlocks/livelocks which are usually the most interesting properties for designers. The technique can also derive the symbolic output of a circuit in terms of its input data and hence the coordination patterns. This can all be done efficiently and with less computational cost than model checking. Furthermore, as our technique is based on symbolic representation of the variables and not the actual values it does not need the data domain to be finite whereas in model checking actual values of variables need to be considered. To the best of our knowledge, this is the first work which uses data constraints in CA for performing the analysis. For doing so, we define the set of regular expressions associated with CA considering data constraints.

Our work is also interesting for the symbolic execution community. We start from automata instead of code, so, we could use the clever way of generating the regular expressions and unfolding the expressions instead of building the symbolic execution tree. We propose a solution for the potentially infinite number of symbolic executions according to the semantics of Reo circuits.

Plan of the paper.

The rest of this paper is organized as follows: Section 2 is a brief overview of Reo and Constraint Automata. In Section 3, we explain our approach to symbolic execution of Reo circuits. Section 4 contains a number of case studies. Related work is presented in Section 5. In Section 6, we discuss future work and conclude the paper.

2 Reo and Constraint Automata

Reo is a model for building component connectors in a compositional manner [1]. It allows modeling the behavior of such connectors, formally reasoning about them, and once proven correct, automatically generating the so-called glue code from the specification. Each connector in Reo is, in turn, constructed compositionally out of simpler connectors, which are ultimately composed out of primitive channels.

A channel is a primitive communication medium with exactly two ends, each with its own unique identity. There are two types of channel ends: source end through which data enters and sink end through which data leaves a channel. A channel must support a certain set of primitive operations, such as I/O, on its ends; beyond that, Reo places no restriction on the behavior of a channel. This allows an open-ended set of different channel types to be used simultaneously together in Reo, each with its own policy for synchronization, buffering, ordering, computation, data retention/loss, etc [1].

Channels are connected to make a circuit. Connecting (or *joining*) channels is putting channel ends together in *nodes*. Thus, a *node* contains a set of channel ends. The semantics of a node depends on its type. Based on the types of its coincident channel ends, a node can have one of three types. If all coincident channel ends on a node are exclusively source (or sink) channel ends, the node is called a source (respectively, sink) node. Otherwise, it is called a mixed node.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a *replicator*. A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends.

Reo offers a set of open ended channels, but a set of primitive channels shown in Figure 1 (with their corresponding CA) are commonly used in Reo circuits. The behavior of every connector in Reo imposes a specific coordination pattern on the entities that perform normal I/O operations through that connector, without the knowledge of those entities. This makes Reo a powerful glue language for a compositional construction of connectors to combine component instances into a software system and exogenously orchestrate their mutual interactions.

2.1 Constraint Automata: Compositional Semantic of Reo

Constraint automata are presented in [3] as a formal semantics for Reo connectors, based on a co-algebraic semantics given in [5]. Using Constraint Automata as an operational model for Reo connectors, the automata states stand for the possible configurations (e.g., the contents of the FIFO-channels of a Reo connector) while the automata-transitions represent the possible data flow and its effect on these configurations. The operational semantics for Reo presented in [1] can be reformulated in terms of Constraint Automata. The Constraint Automaton of a given Reo connector can be derived in a *compositional* way. For this, composition operators for Constraint Automata corresponding to the Reo connector primitives are provided.

A constraint automaton is defined in [3] as follows:

Definition 2.1 [***Constraint Automata***] A constraint automaton (over the data domain *Data*) is a tuple $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ where

- Q is a set of states,
- $\mathcal{N}ames$ is a finite set of names,
- $\longrightarrow$ is a subset of $Q \times 2^{\mathcal{N}ames} \times DC \times Q$, called the transition relation of $\mathcal{A}$, where DC is the set of data constraints,
- $Q_0 \subseteq Q$ is the set of initial states.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in\longrightarrow$. We call N the name-set and g the guard of the transition. For every transition $q \xrightarrow{N,g} p$ we require that (1) $N \neq \emptyset$ and (2) $g \in DC(N, Data)$. $\mathcal{A}$ is called finite iff Q, $\longrightarrow$ and the underlying data domain *Data* are finite.

Data constraints: Data constraints are defined by the following grammar [3]:
$g \;::=\; \mathsf{false} \;\mid\; \mathsf{true} \;\mid\; \varepsilon \;\mid\; d_{<n>} = d \;\mid\; g \vee g \;\mid\; g \wedge g$

Here, n denotes the names and $d \in Data$. We assume a global data domain *Data* for all names. Alternatively, we can assign a data domain $Data_A$ to every name A and require type-consistency in the definition of data constraints. Data constraints (DCs) can be viewed as sets of name-data-assignments. We often use derived DCs such as $d_A \neq d$ or $d_A = d_B$.
The symbol $\models$ stands for the obvious satisfaction relation which results from interpreting DCs over name-data-assignments. Satisfiability and validity, logical equivalence $\equiv$ and logical implication $\leq$ of DCs are defined as usual.

We now explain how Constraint Automata can be used to model the possible data flow in a Reo circuit. To provide a *compositional* semantics for Reo circuits, we need Constraint Automata for each of the basic channel connectors and automata-operations to mimic the behavior of the Reo-operations for join and hiding.

Figure 1 shows the Constraint Automata for the merger node and for the standard basic channel types: A *Sync*, (Figure 1.a) channel has a source(A) and a sink(B) end. It accepts a data item through its source end iff it can simultaneously dispense it through its sink. The channel *SyncDrain* which is shown in Figure 1.b is a channel with two source ends (A and B). It accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. All data accepted by this channel are lost. Figure 1.c is a *LossySync* channel. This channel is also similar to the Sync channel, except that

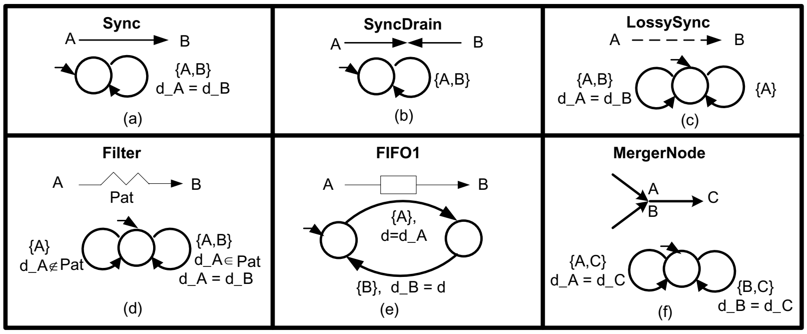


Fig. 1. Reo primitive channels and merger node, and their deterministic Constraint Automata

it always accepts all data items through its source end(A). If it is possible for it to simultaneously dispense the data item through its sink(B) the channel transfers the data item; otherwise the data item is lost. A *Filter* channel which is shown in Figure 1.d behaves like the Sync except that it loses all data that do not match the specified pattern of the filter(*Pat* in the figure). The *FIFO*1 channel(Figure 1.e) has a source(A) and a sink end(B), and a bounded buffer with capacity of 1 data items (the box in the figure). The accepted data items are kept in the internal FIFO buffer of the channel. The appropriate I/O operations on the sink end of the channel obtain the content of the buffer in the FIFO order. A merger node, shown in Figure 1.f, choose the input from one of the channel ends A or B nondeterministically and pass it to the channel end C as the merger output.

The product operator is defined on Constraint Automata that capture the meaning of Reo's join operator [3]. The product-automaton of the two Constraint Automata $\mathcal{A}_1 = (Q_1, \mathcal{N}ames_1, \longrightarrow_1, Q_{0,1})$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}ames_2, \longrightarrow_2, Q_{0,2})$, is [3]:

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 \;=\; (Q_1 \times Q_2, \mathcal{N}ames_1 \cup \mathcal{N}ames_2, \longrightarrow, Q_{0,1} \times Q_{0,2})$$

where $\longrightarrow$ is defined by the following rules:

$$\frac{q_1 \xrightarrow{N_1,g_1}_1 p_1, \quad q_2 \xrightarrow{N_2,g_2}_2 p_2, \quad N_1 \cap \mathcal{N}ames_2 = N_2 \cap \mathcal{N}ames_1}{\langle q_1, q_2\rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2\rangle}$$

and

$$\frac{q_1 \xrightarrow{N,g}_1 p_1, \quad N \cap \mathcal{N}ames_2 = \emptyset}{\langle q_1, q_2\rangle \xrightarrow{N,g} \langle p_1, q_2\rangle}$$

and latter's symmetric rule. The first rule is applied when in the automata there are two transitions which can be fired together. This happens only if there is no shared name in both automata which is present on one of the transitions but not present on the other one. In this case the transition in the resulting automaton has the union of the names on both transitions, the data constraint is the conjunction of the data constraints of the two transitions. The second rule is applied when a transition in one automaton can be fired independently of the other automaton, which happens when the names on the transition are not included in the other automaton. After a join operation, hiding in the result automaton can be done. Hiding abstracts the details of internal communication among channels in a Reo circuit, and shows the observable behavior of a Reo circuit.

An extension of Constraint Automata with State Memory(CASM), explicitly partitions $\mathcal{N}ames$, into three sets of $\mathcal{N}ames^{src}$, $\mathcal{N}ames^{snk}$, and $\mathcal{N}ames^{mix}$ and extends data constraints to accommodate state memory cells [2]. Here, we use a slightly simplified form of the CASM, using the state memories and leave out the separation of the name set. The formal definition is as follows:

Definition 2.2 (*Constraint Automata with State Memory (CASM)*). A constraint automaton with state memory is a tuple $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0, \mathcal{M}, \mathcal{V}_0)$ where [2]:

- Q, $\mathcal{N}ames$, $\longrightarrow$, and Q_0 are defined the same as for ordinary CA.
- $\mathcal{M}$ is a set of memory cells partitioned into memory cells $\mathcal{M}_q$, for all $q \in Q$.
- For every $q \in Q$ the value function $\mathcal{V}_q : M_q \to Data$ is defined when the automaton is in state q. The set $\mathcal{V}_0$ consists of the initial value function $\mathcal{V}_{q_0}$, each of which gives the initial values of its corresponding initial state $q_0 \in Q_0$.
- The data constraint language is extended to include relativized memory cell names of the source and target state of the transition ($s.m$ and $t.m$, for $m \in \mathcal{M}$), as well as the usual d_n for node names $n \in \mathcal{N}ames$, such that for each transition $q \xrightarrow{N,g} p$, the free variables of g are in the set $\{s.m|m \in \mathcal{M}_q\} \cup \{t.m|m \in \mathcal{M}_p\} \cup \{d_n|n \in \mathcal{N}ames\}$.
- Given a data assignment $\delta_N : N \to Data$ and a value function $\mathcal{V}_q : \mathcal{M}_q \to Data$, a transition $q \xrightarrow{N,g} p$ is possible only if there exists a value function $\mathcal{V}_p : \mathcal{M}_p \to Data$ such that g is true under the mappings δ_N, $\mathcal{V}_p$, and $\mathcal{V}_q$. Firing the transition, makes $\mathcal{V}_p$ the value function for the memory cells $\mathcal{M}_p$ of state p.

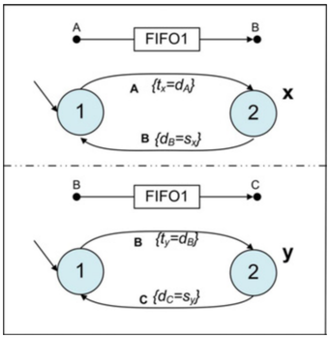
Data constraints are written according to the following grammar:

$$\begin{aligned} g \ ::= \ & \mathsf{false} \ | \ \mathsf{true} \ | \ \varepsilon \ | \ d_{<n>} = d \ | \ g \vee g \ | \ g \wedge g \ | \\ & t_{<m>} = d_{<n>} \ | \ d_{<n>} = s_{<m>} \ | \ d_{<n>} = d_{<n>} \ | \ d_{<n>} = v \end{aligned}$$

Where m ranges over memory cells, n ranges over $\mathcal{N}ames$, and v ranges over the data set $Data$.

The special symbols s and t refer to, respectively, the source and the target states of a transition: the names s_x and t_x in the data constraint g of a transition $q \xrightarrow{N,g} p$ refer to memory cells $q.x$ and $p.x$, respectively. Thus, data constraints of transitions can refer to state memory cells only indirectly, through the relative references of the form s_x or t_x (as opposed to the explicit memory cell names $q.x$ or $p.x$). This has two advantages. First, it makes it impossible for a transition to refer to a memory cell of any state other than its own source or target. Second, it simplifies the product operation by eliminating the need to do anything special (e.g., name substitution) when we combine synchronous transitions.

We can see the CASM for two FIFO1 (FIFO with buffer capacity 1) channels



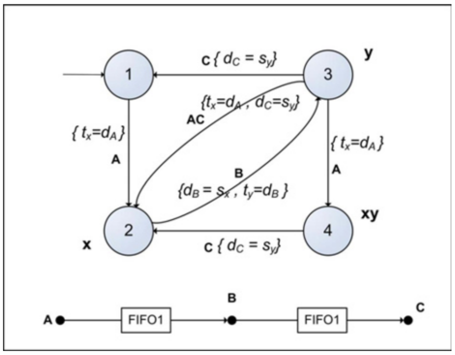


Fig. 2. Constraint Automata with state memory for (a) FIFO1 and (b) FIFO2

in Figure 2(a). In this figure x and y are the internal buffers of the two FIFO1 channels. Joining two FIFO1 channels, putting the sink end of one on the same node as the source end of the other (in this figure node B), gives us a FIFO2 (FIFO with buffer capacity 2) channel as in Figure 2(b). The product of the Constraint Automata of two channels gives us the constraint automaton of a FIFO2 channel. In this example, we see how the data constraints on the value of the memory cells of each state are referred to as the target and the source of a transition; and how data passing through the buffers of a Reo circuit and correspondingly through the states of CASM is depicted transitively.

3 Symbolic Execution of Reo Circuits

The main idea behind symbolic execution is to use symbolic values, instead of actual data, as input values and to represent the values of program variables as symbolic expressions. Consequently, the output values computed by a program are expressed as a function of the input symbolic values.

Symbolic execution is a natural extension of normal execution, rendering normal computation as a special case [12]. In the case of conventional imperative programming languages, definitions for the basic operators of the language are extended to accept symbolic input values and produce symbolic formulas as output. For instance, the expression on the right-hand side of an assignment statement is evaluated, possibly substituting polynomial expressions for variables. The result is a polynomial (an integer is the trivial case) which is then assigned as the new value of the variable on the left-hand side of the assignment statement.

The state of a symbolically executed program includes the symbolic values of program variables, the program counter, and a path condition. The path condition is a quantifier-free Boolean formula over the symbolic input values; it accumulates constraints that the input values must satisfy in order for an execution to follow its particular associated path. A symbolic execution tree characterizes the execution paths followed during the symbolic execution of a program. The nodes in this tree represent program states and the arcs represent transitions between states [18].

3.1 CA, TDS, and Symbolic Execution of Reo

The coalgebraic formal semantics for Reo connectors, presented in [5], assigns to any Reo connector a relation over infinite timed data streams, called its TDS language. This language can be used in symbolic execution of Reo and in obtaining a symbolic relation between input and output values of a circuit. To reason about TDS languages, we may regard Constraint Automata as acceptors for sets of timed data streams.

As shown in Section 2, the transitions of Constraint Automata are labeled with pairs consisting of a nonempty subset $N = \{A_1, ..., A_n\}$ of nodes and a data constraint g. Data constraints can be viewed as a *symbolic representation* of sets of data assignments. A data assignment is a function from the name set N to the data domain $Data$. The notation $d_A = d$ shows the assignment of $d \in Data$ to the name $A \in N$. Formally, data constraints are propositional formulae built from the atoms $d_A = d$, where data item d is assigned to port A. In all primitive Reo channels that we consider in this paper, we abstract from concrete values of d. We have only equalities (and not assignments) among the data elements passing through the ports. Also, the only Boolean connector used is the *and* connector. The primitive Reo channels (and hence their products) satisfy this property. For simplicity, we consider only deterministic CA with single initial states.

Our approach. In CA, instead of specific data values, we have symbolic representation of input and output values and transitions show the possible relations among them. This makes the CA an appropriate basis to build a symbolic execution tool. Here, we use the data constraints for deriving the relation between input and output values.

The problem is to obtain the possible relations between output and input values, represented symbolically. For that, we need to obtain the execution paths in the symbolic execution tree of a given Reo circuit using its constraint automaton. The number of execution paths in a symbolic execution tree is infinite if cycles are present in the CA of the circuit. Consider the memory cells in the cycle that are in a relation by appearing in the same data constraint. Let k be the length of the longest path between memory cells in their transitive relation. By traversing each cycle k times, we can generate the expression denoting the relation between the last k elements of the data streams, where k is a bounded integer. For each data stream, entering/exiting each port, k may be different and depends on the number of elements that are buffered in the circuit. This also shows whether an output depends on an input. If we cannot find the expected relation between certain nodes in $Names$, this indicates a possible bug, i.e., a mismatch between the specification of the problem and our Reo circuit.

In our technique, instead of building the symbolic execution tree, we generate the regular expression of a CA assuming its initial state as the final state of the automaton. Execution paths are derivatives of regular expressions. We unfold data constraints and show the relation among the data elements in the data streams. The transitive closure over this relation gives us the relations among symbolic input and output values.

Conventional symbolic execution techniques start by building the symbolic execution tree. To make our technique more approachable and to show the intuition, in the following, we first show how a symbolic execution tree of a Reo circuit is formed. But, instead of a program code here we already have the Constraint Automata. So, after showing the execution tree we explain our technique using the regular expression of a CA for constructing the required execution paths and deriving the relations among input and output values.

3.2 Symbolic Execution Tree of a Reo Circuit

The symbolic execution tree of a given Reo circuit is formed by traversing the constraint automaton of the circuit. We start from the initial state in the CA and walk through all possible paths in the CA. While traversing a transition $q \xrightarrow{N,g} p$ of the CA, we store its name-set N and its guard g on its respective transition in the traversal tree.

As we construct the tree, every time we see a name (port in Reo) it means that a new data element is observed in the stream of data passing through that port. The data stream passing through a port A is denoted as:

$a = (a_0, a_1, a_2, a_3, ..., a_{last-2}, a_{last-1}, a_{last})$

While writing the data constraints for each state in the traversal tree, we use the last elements of these data streams: $..., a_{last-2}, a_{last-1}, a_{last}$ for each port: unfolding the data stream while keeping the symbolic representation and the relation among the data elements. Figure 3 shows the execution tree for a FIFO2 channel. In this figure the relations among data elements are shown. All execution paths from the initial state back to the initial state are generated. The string on each leaf of the tree shows the names of the ports that fire in its corresponding path. A pair of square brackets encloses a set of port names that fire synchronously (see Step 1 in Section 3.3). The boxes connected to each node of the tree show the names that fire to reach that node (CA state) together with all the data constraints that hold at that point (have been met until that point), and the transitive relations among them.

We consider the set of data constraints (Boolean expressions for the data values) as a relation on the data elements passing through the nodes and the data elements that are buffered in the memory cells (where we have FIFO channels). We consider primitive Reo channels only, which yield only equality in the data constraints. Hence, the relations corresponding to the data passing through the nodes are symmetric. Relations between memory cells and data elements must be resolved correctly. The transitive closure of this relation gives us all symbolic relations among input and output values (ports).

The symbolic execution tree of Figure 3 is generated by traversing each cycle in the CA only once. In this example, this is enough to yield the general expression for the relation between input and output values. No new relation will be generated by traversing the cycles more than once. In some cases we may be interested only in finding the relation between the data passing through a certain output node and

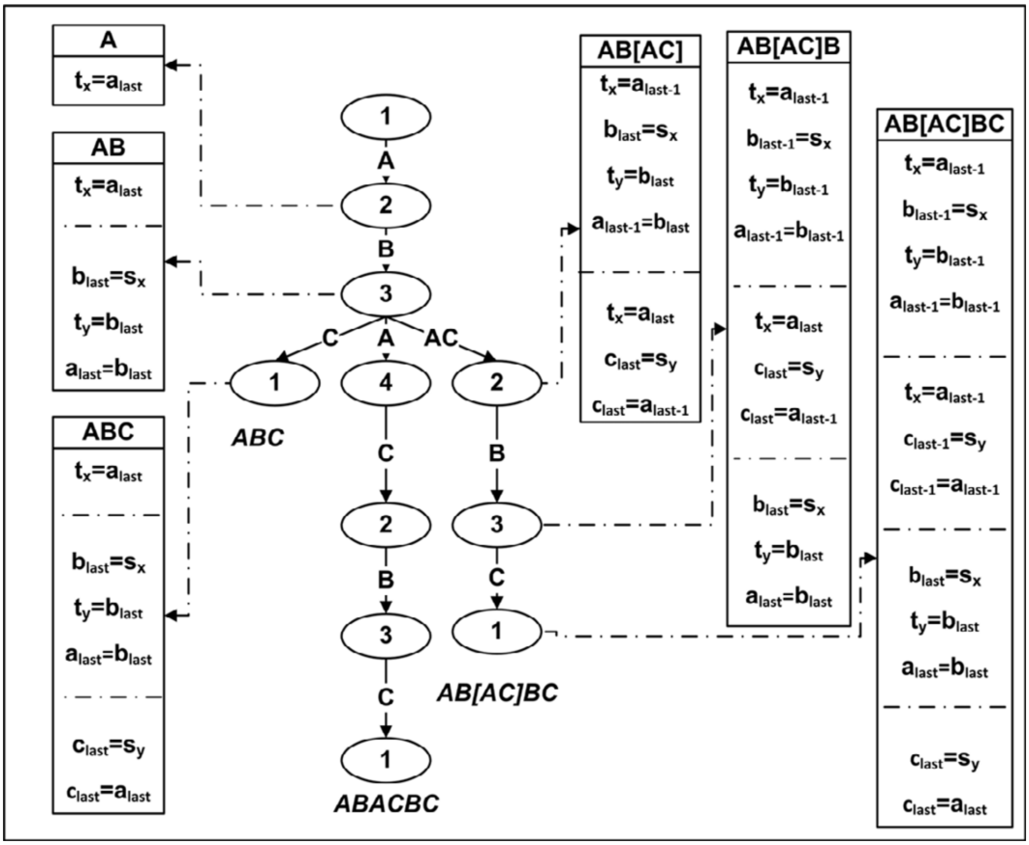


Fig. 3. Symbolic execution tree for FIFO2: each node represents a state and edges represent transitions in CA (string on each leaf: names of the ports that are fired in the corresponding path; a pair of square brackets: encloses a set of port names that are fired synchronously).

the input data. In this case, we focus only on the paths including the name of this particular output node.

Symbolic execution has been used for verification of programs, and the techniques are naturally based on building the symbolic execution tree. But in our case we are a step ahead and have the Constraint Automata instead of a the source code of program. Thus, instead of building the symbolic execution tree, we use the regular expression of the CA to obtain the set of derivatives of our interest. The purpose of presenting the symbolic execution tree here is to show the similarities of our technique with the conventional symbolic execution methods.

3.3 Symbolic Outputs of Reo Circuits

We compute the symbolic output values of a Reo circuit in three steps.

- (1) Obtain the regular expression for the constraint automaton. Here, we use Brzozowski's algebraic method [9]. We consider the initial state as the final state of the CA.
- (2) Construct the derivatives (words) of the regular expression, by substituting

the Kleene closure $'*'$ with k or zero repetition where k is the longest path between memory cells in the cycle in their transitive relation. These derivatives represent the set of all execution paths in the constraint automaton (leaves in the symbolic execution tree) if we go through each cycle of the CA k times (substitution of $'*'$ with k), or bypass the cycle where possible (substitution of $'*'$ with zero).

- (3) Build the symbolic data stream of each output node based on the symbolic input stream of data for each derivative, by forming the transitive closure of the relation among the elements of data in each stream.

  In this step, we traverse each derivative from right to left and substitute data elements of data streams in the formula on data constraints (add proper indices for each element). Then, we compute the transitive closure of relations between data elements in input and output data streams.

In the following, we explain each step and use the FIFO2 channel shown in Figure 2(b) as the running example.

Step 1: Produce Regular Expression. We use Brzozowski algebraic method [9] and Arden's theorem [6] for generating regular expressions for Constraint Automata. We create a system of regular expressions with one regular expression as the unknown for each state in a constraint automaton, and then we solve the system for R_1 where R_1 is the regular expression associated with the initial state q_1. For our purposes, we consider initial states of Constraint Automata as final states because when we get back to these states we are back in the initial configuration of the Reo circuit and we can assume this as completing a run in the Reo circuit. We also consider the data constraints for each transition of a constraint automaton. We keep the data constraint and the transition labels (names) as a pair. Note that in applying Brzozowski algebraic method we keep the pair of names and the data constraints for each transition and carry them while solving the system.

In a constraint automaton we may have more than one name on a transition, which means that all Reo ports corresponding to these names fire synchronously on that transition. We put these names in square brackets, [], to show their synchronous/atomic firing. For example, $[ABC]$ means that A, B, and C fire atomically in any order (even simultaneously together) on one transition. In contrast, (ABC) means that A, B, and C fire one after the other on successive transitions. As such, $[ABC]$ and $[CAB]$ are identical to one another (and to the rest of the permutations of this set of names), while (ABC) and (CAB) are clearly different. So, the alphabet of our language consists of composite letters. The first component of a letter can be a name from the *Name* set of a CA or a bracket containing a (nonempty) set of names. The second component is the data constraint of the corresponding transition.

The regular expression for a FIFO1 channel in Figure 2(a) is: $R_1 = (AB)^*$, augmenting it with data constraints we have $R_1 = (A\{t_x = d_A\}B\{d_B = s_x\})^*$. For our running example, the FIFO2 of Figure 2(b), we have $R_1 = ((A)(B[AC] + BAC)^*BC)^*$
and the complete expression including the data constraints is:

$R_1 = ((A\{t_x = d_A\})(B\{d_B = s_x, t_y = d_B\}[AC]\{t_x = d_A, d_C = s_y\} + B\{d_B = s_x, t_y = d_B\}A\{t_x = d_A\}C\{d_C = s_y\})^* B\{d_B = s_x, t_y = d_B\}C\{d_C = s_y\})^*$

For *LossySync*, we add the data constraint $\{d_A = d_A\}$ on the transition with the name A to show losing of data.

Step 2: Produce all derivatives for the regular expression by substituting each $'*'$ with zero or k repetition. In this step, we generate all possible derivatives of the regular expression by considering the repetition in the expression to be zero or k (where k is the length of the longest path between memory cells in the cycle in their transitive relation) . This means that in the case of cycles in the CA, we traverse the cycle only k times.

Below, we show the derivatives of the regular expression by substituting each $'*'$ (repetitions) with zero or $k = 1$ for the FIFO2 example:

- Substituting the outer $'*'$ with zero repetition:
  $R_{1_1} = null$

- Substituting the inner $'*'$ with zero repetition and the outer $'*'$ with one:
  $R_{1_2} = (A\{t_x = d_A\}B\{d_B = s_x, t_y = d_B\}C\{d_C = s_y\})$

- Substituting the inner $'*'$ with one repetition and the outer $'*'$ with one:
  $R_{1_3} = ((A\{t_x = d_A\})(B\{d_B = s_x, t_y = d_B\}[AC]\{t_x = d_A, d_C = s_y\} + B\{d_B = s_x, t_y = d_B\}A\{t_x = d_A\}C\{d_C = s_y\})B\{d_B = s_x, t_y = d_B\}C\{d_C = s_y\})$

Step 3: Build the symbolic data stream of each output node based on the symbolic input stream of data for each derivative. In this step we traverse each regular expression to show the data stream by its elements (indexing the elements of data stream). Then, obtaining all transitive relations among these elements of streams (transitive closure), gives us the relation between output and input values. In this step, we first specify indices for each data element in the data constraints shown in the regular expression. To do this, we traverse the regular expression from right to left. Then, we obtain the transitive closure.

Step 3.1: Indexing. Intuitively, the purpose of indices is to show the order of the nodes seen along the traversal of a regular expression. For example, if we observe a name A for the first time in a regular expression, we replace its data d_A with the indexed name a_{last}. If we see another A we denote the new one as a_{last}, rename the previous last one as a_{last-1}, and the older ones as a_{last-2}, and so on.

For the FIFO2 example, we obtain the following equation for the second derivative in step 2:

$R_{1_2} = (A\{t_x = a_{last}\}B\{b_{last} = s_x, t_y = b_{last}\}C\{c_{last} = s_y\})$

Note that in indexing the data elements from right to left, the starting index for all subexpressions of a $'+'$ is the same. The equation for the last regular expression in step 2 is:

$R_{1_3} = ((A\{t_x = a_{last-1}\})(B\{b_{last-1} = s_x, t_y = b_{last-1}\}[AC]\{t_x = a_{last}, c_{last-1} = s_y\} + B\{b_{last-1} = s_x, t_y = b_{last-1}\}A\{t_x = a_{last}\}C\{c_{last-1} = s_y\})B\{b_{last} = s_x, t_y = b_{last}\}C\{c_{last} = s_y\})$

Double Indexing. When we finish traversing a parenthesized expression for indexing, we may end up with two different indexed versions of a Reo node name. This may happen if the expression contains at least one $'+'$ operator. In this case, we continue indexing for each version, and then compose both indexed versions of the name for the data elements with the logical $\vee$ operator. For example, in $R = (A\{t_x = d_A\}(AB\{d_B = d_A\} + B\{d_B = s_x\}))$ we compute the indices for A (and B) as follows: $R = (A\{t_x = a_{last \vee last-1}\}(AB\{b_{last} = a_{last}\} + B\{b_{last} = s_x\}))$.

Step 3.2: Transitive Closure. Having properly indexed the data elements, we derive all transitive relations among them. For our FIFO2 example, we have

$$R_{1_2} = (A\{t_x = a_{last}\}B\{b_{last} = s_x, t_y = b_{last}\}C\{c_{last} = s_y\})$$

As we know the relation between each t_x and s_x, we can write:

$$R_{1_2} = (AB\{t_x = a_{last}, b_{last} = s_x, t_y = b_{last}, a_{last} = b_{last}, t_y = a_{last}\}C\{c_{last} = s_y\})$$

And then:

$R_{1_2} = (ABC\{t_x = a_{last}, b_{last} = s_x, t_y = b_{last}, a_{last} = b_{last}, t_y = a_{last}, c_{last} = s_y, c_{last} = a_{last}\})$

Now, focusing only on the relation between the output node C and the input node A (which is the relation we are interested in) we have:

$$R_{1_2} = (ABC\{c_{last} = a_{last}\})$$

Now we can conclude that the data of c_i is the same as the data of a_i. The pattern ABC in the above equation shows that the data of c_i is exactly the data of a_i with a possible delay. The situation would be different if the pattern were $[ABC]$, since this implies that the firing is atomic.

We continue for R_{1_3}:

$R_{1_3} = ((A\{t_x = a_{last-1}\})(B\{b_{last-1} = s_x, t_y = b_{last-1}\}[AC]\{t_x = a_{last}, c_{last-1} = s_y\} + B\{b_{last-1} = s_x, t_y = b_{last-1}\}A\{t_x = a_{last}\}C\{c_{last-1} = s_y\})B\{b_{last} = s_x, t_y = b_{last}\}C\{c_{last} = s_y\})$ Then:

$$R_{1_3} = (A(B[AC] + BAC)BC\{a_{last-1} = c_{last-1}, c_{last} = a_{last}\})$$

This gives us the relation: $c_{last} = a_{last}, c_{last-1} = a_{last-1}$, meaning that we have two elements of a in the Reo circuit during one run of the circuit. Each element of c in the output node is the same as the element of a in the input node of the Reo circuit.

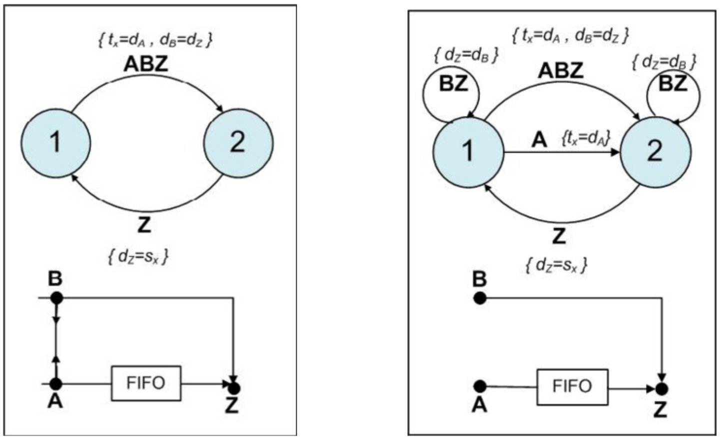


Fig. 4. Circuits (a) Alternater and (b) Non-Deterministic Choice, and their Constraint Automata

In the end, if we can describe the data stream on an output Reo node, say Z, by a set of input data streams S, then we have achieved our goal. Otherwise, the Reo node Z is unreachable from the Reo nodes in the set S, and if S is the set of all input values then we have discovered a *deadlock* or *livelock* trap.

4 Case Studies

In this section we present a few examples.

Example 1: Alternator circuit. The alternator circuit is shown in Figure 4(a). The steps for generating the regular expression of this circuit, generating its derivatives, indexing, and deriving the relation among its input and output values are included in the Appendix.

The final result is:

$$z_{last-1} = b_{last}, z_{last} = a_{last}$$

This shows that (1) the output data from port Z are formed by alternating between the input data from ports A and B, and (2) the data from A is synchronously produced on Z, whereas the data from B is produced on Z with a one cycle delay.

Example 2: Non-Deterministic Choice. Figure 4(b) shows a Reo circuit with a non-deterministic choice between its input ports. The steps for generating the regular expression of the circuit, generating its derivatives, indexing, and deriving the relation among its input and output values are included in the Appendix. The final result is:

$$z_{last-1} = b_{last}, z_{last} = a_{last} \vee z_{last} = b_{last} \vee z_{last} = a_{last}$$

This shows that all three relations between the input and the output values are possible. We have a non-deterministic behavior here because of the merger, which

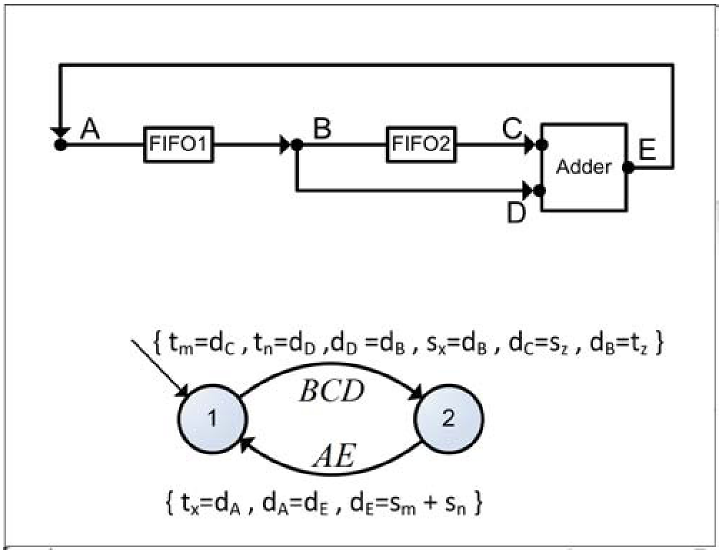

A
FIFO1
B
FIFO2
C
Adder
E
D
{ tm=dC , tn=dD ,dD =dB , sx=dB , dC=sz , dB=tz }
BCD
1
2
AE
{ tx=dA , dA=dE , dE=sm + sn }

Fig. 5. Fibonacci Reo circuit and its constraint automaton.

leads to an expression that represents the general relation among the input and output values of this circuit.

Example 3: Fibonacci series. The Reo circuit for the Fibonacci series [7] has a feedback from its output to its input to generate values of this recursive formula. Figure 5 shows a Fibonacci Reo circuit with its constraint automaton. In this circuit FIFO1 has the initial value of one and FIFO2 has the initial value of zero. Here, we are interested in the relation between an output feeding back as an input to the same circuit. We may notice this feedback when in the regular expression we see the same node as both source and target in a recurring data constraint. For extracting the recurring relation, we need to traverse each cycle a finite number of times. In this example, x is the memory cell for FIFO1, y and z are memory cells for FIFO2, and m and n are memory cells for the Adder component. The Adder component is just a simple component that gets its two input values, adds them, and puts them on its output port. The steps for generating the regular expression of the circuit, generating its derivatives, indexing, and deriving the relation among its input and output values are included in the Appendix. In this example, because of the feedback, we are interested in deriving the relation between the sequence of data on node E of the Reo circuit with itself. So, we repeat the cycle and the indexing until we have a closed expression of data values, i.e., we have data on E specified in terms of itself (here it is three times which is equal to the total number of buffers). The final result gives us the following symbolic output for the Fibonacci circuit:

$$e_{last} = e_{last-1} + e_{last-2}$$

Example 4: Critical Section. In [19] the behavior of a CPU is modeled by defining a set of basic components for arithmetic, logic, and comparison operations, and Reo circuits are used for branch, move, and basic I/O instructions. It is shown how these primitive operations can be structured using Reo. This approach can be used for analyzing multi-thread systems, database applications and resource sharing systems. We pick the case study of [19] as an example of a control-dominated system. It is a system consisting of two concurrent processes with a critical section that is modeled using Reo and CA. In the model for this system, there are 84 states, 284 transitions, and 127 names in the CA. For each process, an output is generated after it passes the critical section. We checked in our symbolic execution result whether this output is generated or not. We found an execution path in this system in which this output was not generated. This showed that a deadlock had occurred. Thus, we could detect a deadlock in the system without generating the whole state space, which is hence less expensive than traditional model checking.

5 Related Work

Symbolic execution is an established technique in verification, especially in the hardware verification community. Having its roots in the seventies [12], symbolic execution rapidly grew to be one of the predominant verification technique for hardware designs, with many successful applications, such as [16]. With the invention of symbolic trajectory evaluation by Seger and Bryant in the nineties [8] this approach has now become a component in most of the industrial hardware verification systems (e.g. FORTE [22] at Intel). As Reo has previously been used for modeling hardware systems [20,21], our approach in this paper can be compared with the previous papers on symbolic execution of hardware systems, in the sense that we are also able to symbolize the relation between input and output values in a hardware design.

Although not as popular as it is for hardware, symbolic execution is used as a verification technique for software systems as well. The degree of non-determinism and the large number of different factors that determine the result of a software system usually prevent symbolic execution to be used as the only verification technique in practice without any other accompanying technique. Examples includes [14] which converts a concurrent Ada program to a type of Petri net, and then uses symbolic execution to find the relation between the input and the output values of the program. This work has similarities with our work, because it attempts to symbolically analyze a model that is used for representing concurrent behavior.

Symbolic execution is mostly used for generating input test sequences in the common practice of software engineering. In [11] its authors devise a framework on top of a model checker (Java PathFinder) to generate test inputs for Java programs. This framework has been extended in [18]. The main concern in this work is handling complicated data structures and loops effectively. In [17], the authors describe an approach to testing complex safety critical software that combines unit-level symbolic execution and system-level concrete execution for generating test cases that satisfy user-specified testing criteria. They applied their approach to testing a prototype NASA flight software component and discovered a serious bug.

The work in [10] shows how to automatically find bugs in malicious file system code using symbolic execution. Rather than running the code on manually-constructed concrete input, they instead run it on symbolic input. They checked the disk mounting code of three widely-used Linux file systems and found bugs in all of them where malicious data could either cause a kernel panic or form the basis of a buffer overflow attack.

In [23] a method for verifying the correctness of parallel programs is presented. They check the equivalency of the parallel program and a sequential version of it which serves as the specification for the parallel one. They use model checking, together with symbolic execution, to establish the equivalence of the two programs. In this approach the path condition from symbolic execution of the sequential program is used to constrain the search through the parallel program.

6 Conclusion and Future Work

We have developed an approach and a simple tool for symbolic execution of Reo circuits. We use the regular expression corresponding to the CA of a Reo circuit to derive a set of execution paths for the circuit. These execution paths are generated by traversing the CA from its initial state back to its initial state, passing each cycle only a finite number of times (depending on the relations between memory cells which can be obtained by static checking). In the regular expression we consider the data constraints as well as the names of the ports that fire in each transition. This helps us to obtain the transitive relation between the input and the output values, which is represented in a symbolic form.

In our ongoing research, we are currently working on obtaining the general expression for the set of all input and output streams of a Reo circuit. We are also working on different applications amenable to our symbolic technique. We will use compositionality techniques in our symbolic execution approach.

Acknowledgement

The work of the second author is partly supported by CWI (within the project "Synthesis and Analysis of Connector Components (SYANCO)" DN62-613).

References

[1] Arbab, F., *Reo: A channel-based coordination model for component composition*, Mathematical Structures in Computer Science **14** (2004), pp. 329–366.

[2] Arbab, F., *CASM: Constraint automata with state memory*, unpublished notes (2007).

[3] Arbab, F., C. Baier, J. J. Rutten and M. Sirjani, *Modeling component connectors in Reo by constraint automata (extended abstract)*, in: *Proceedings of Second International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03)*, Electronic Notes in Theoretical Computer Science **97** (2004), pp. 25–46.

[4] Arbab, F., N. Medvidovic, N. Mehta and M. Sirjani, *Modeling behavior in compositions of software architectural primitives*, in: *ASE*, 2004, pp. 371–374.

[5] Arbab, F. and J. Rutten, *A coinductive calculus of component connectors*, in: *WADT'02*, LNCS **2755**, 2002, pp. 34–55.

[6] Arden, D. N., *Delayed-logic and finite-state machines*, Theory of Computing Machine Design (1960), pp. 1–3.

[7] Baier, C., M. Sirjani, F. Arbab and J. J. Rutten, *Modeling component connectors in Reo by constraint automata*, Science of Computer Programming **61** (2006), pp. 75–113.

[8] Bryant, R. E., D. L. Beatty and C.-J. H. Seger, *Formal hardware verification by symbolic ternary trajectory evaluation*, in: *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation* (1991), pp. 397–402.

[9] Brzozowski, J. A., *Derivatives of regular expressions*, Journal of ACM **11** (1964), pp. 481–494.

[10] Junfeng Yang, P. T. C. C., Can Sar and D. Engler, *Automatically generating malicious disks using symbolic execution*, in: *Proceedings of the 2006 IEEE Symposium on Security and Privacy table of contents* (2006), pp. 243 – 257.

[11] Khurshid, S., C. S. Pasareanu and W. Visser, *Generalized symbolic execution for model checking and testing*, in: *Proceedings of TACAS'03*, 2003, pp. 553–568.

[12] King, J. C., *Symbolic execution and program testing*, Communications of the ACM **19** (1976), pp. 385–394.

[13] Klüppelholz, S. and C. Baier, *Symbolic model checking for channel-based component connectors*, in: *FOCLASA'06*, 2006.

[14] Morasca, S. and M. Pezzè, *Validation of concurrent ada programs using symbolic execution*, in: *ESEC '89: Proceedings of the 2nd European Software Engineering Conference* (1989), pp. 469–486.

[15] Mousavi, M. R., M. Sirjani and F. Arbab, *Formal semantics and analysis of component connectors in Reo*, in: *FOCLASA'05*, Electronic Notes in Theoretical Computer Science **154**, 2006, pp. 83–99.

[16] Pandey, M., R. Raimi, R. E. Bryant and M. S. Abadir, *Formal verification of content addressable memories using symbolic trajectory evaluation*, in: *DAC '97: Proceedings of the 34th annual conference on Design automation* (1997), pp. 167–172.

[17] Pasareanu, C. S., P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person and M. Pape, *Combining unit-level symbolic execution and system-level concrete execution for testing nasa software*, in: *Proceedings the international symposium on Software testing and analysis* (2008), pp. 15–26.

[18] Pasareanu, C. S. and W. Visser, *Verification of java programs using symbolic execution and invariant generation*, in: *Proceedings of SPIN'04*, 2004, pp. 164–181.

[19] Pourvatan, B., A. Afshar and N. Rouhy, *Modeling sequential and concurrent programs with Reo and constraint automata*, Technical report (2007).

[20] Razavi, N. and M. Sirjani, *Using Reo for formal specification and verification of system designs*, in: *Proceedings of MEMOCODE 2006* (2006), pp. 113–122.

[21] Razavi, N. and M. Sirjani, *Compositional semantics of system-level designs written in SystemC*, in: *Proceedings of FSEN 2007* (2007), pp. 113–128.

[22] Seger, C.-J. H., R. B. Jones, J. W. O'Leary, T. F. Melham, M. Aagaard, C. Barrett and D. Syme, *An industrially effective environment for formal hardware verification*, IEEE Trans. on CAD of Integrated Circuits and Systems **24** (2005), pp. 1381–1405.

[23] Siegel, S. F., A. Mironova, G. S. Avrunin and L. A. Clarke, *Combining symbolic execution with model checking to verify parallel numerical programs*, ACM Transactions on Software Engineering and Methodology **17** (2008), pp. Article 10, 1–34.

[24] Tasharofi, S. and M. Sirjani, *Formal modeling and conformance validation for WS-CDL using Reo and CASM*, in: *Proceedings of FOCLASA08*, Electronic Notes in Theoretical Computer Science **159** (2008), pp. 99 – 115.

[25] Tasharofi, S., M. Vakilian, R. Ziloochian and M. Sirjani, *Modeling web services using coordination language Reo*, in: *Proceedings of WS-FM07*, Lecture Notes in Computer Science **4937** (2007), pp. 108 – 123.

Appendix: Deriving Symbolic Outputs for the Case studies

Example 1: Alternator circuit. The steps for generating the regular expression of this circuit, generating its derivatives, indexing, and deriving the relation among its input and output values are as follow:
The regular expression: $R_1 = ([ABZ]\{t_x = d_A, d_B = d_Z\}Z\{d_Z = s_x\})^*$
The regular expression derivative: $R_1 = ([ABZ]\{t_x = d_A, d_B = d_Z\}Z\{d_Z = s_x\})$
It is indexed as: $R_1 = ([ABZ]\{t_x = a_{last}, b_{last} = z_{last-1}\}Z\{z_{last} = s_x\})$
Forming the transitive closure of the data constraints, we have:

$$R_1 = ([ABZ]Z\{t_x = a_{last}, b_{last} = z_{last-1}, s_x = z_{last}, z_{last} = a_{last}\})$$

And finally we have:

$$z_{last-1} = b_{last}, z_{last} = a_{last}$$

Example 2: Non-Deterministic Choice. The steps for generating the regular expression of the circuit, generating its derivatives, indexing, and deriving the relation among its input and output values are as follow:
The regular expression:
$R_1 = ([BZ]\{d_Z = d_B\} + ([ABZ]\{t_x = d_A, d_B = d_Z\} + A\{t_x = d_A\})([BZ]\{d_Z = d_B\})^*Z\{d_Z = s_x\})^*$

The regular expression derivatives:
$R_{1_1} = ([BZ]\{d_Z = d_B\} + ([ABZ]\{t_x = d_A, d_B = d_Z\} + A\{t_x = d_A\})([BZ]\{d_Z = d_B\})Z\{d_Z = s_x\})$
$R_{1_2} = ([BZ]\{d_Z = d_B\} + ([ABZ]\{t_x = d_A, d_B = d_Z\} + A\{t_x = d_A\})Z\{d_Z = s_x\})$

They are indexed as:
$R_{1_1} = ([BZ]\{z_{last} = b_{last}\} + ([ABZ]\{t_x = a_{last}, b_{last-2} = z_{last-2}\} + A\{t_x = a_{last}\})([BZ]\{z_{last-1} = b_{last}\})Z\{z_{last} = s_x\})$
$R_{1_2} = ([BZ]\{z_{last} = b_{last}\} + ([ABZ]\{t_x = a_{last}, b_{last} = z_{last-1}\} + A\{t_x = a_{last}\})Z\{z_{last} = s_x\})$

Forming the transitive closure of the data constraints, we have:
$R_{1_1} = ([BZ]\{z_{last} = b_{last}\} + ([ABZ]\{t_x = a_{last}, b_{last-1} = z_{last-2}\} + A\{t_x = a_{last}\})([BZ]\{z_{last-1} = b_{last}\})Z\{z_{last} = s_x\})$
$R_{1_2} = ([BZ]\{z_{last} = b_{last}\} + ([ABZ]\{t_x = a_{last}, b_{last} = z_{last-1}\} + A\{t_x = a_{last}\})Z\{z_{last} = s_x\})$

And finally we have:

$$z_{last-1} = b_{last}, z_{last} = a_{last} \vee z_{last} = b_{last} \vee z_{last} = a_{last}$$

Example 3: Fibonacci series. The steps for generating the regular expression of the circuit, generating its derivatives, indexing, and deriving the relation among its input and output values are as follow:
The regular expression:
$R_1 = ([BCD]\{t_m = d_C, t_n = d_D, d_D = d_B, d_B = s_x, d_C = s_z, t_z = d_B\}[AE]\{t_x = d_A, d_A = d_E, d_E = s_m + s_n\})^*$

The regular expression derivative:
$R_1 = [BCD]\{t_m = d_C, t_n = d_D, d_D = d_B, d_B = s_x, d_C = s_z, t_z = d_B\}[AE]\{t_x = d_A, d_A = d_E, d_E = s_m + s_n\}$

In this example, because of the feedback, we are interested in deriving the relation between the sequence of data on node E of the Reo circuit with itself. So, instead of going through the cycle once, we repeat the cycle and the indexing until we have data on E being specified by itself (here it is three times which is equal to the total number of buffers). So, we have:
$R_1 = [BCD]\{t_m = d_C, t_n = d_D, d_D = d_B, d_B = s_x, d_C = s_z, t_z = d_B\}[AE]\{t_x = d_A, d_A = d_E, d_E = s_m + s_n\}[BCD]\{t_m = d_C, t_n = d_D, d_D = d_B, d_B = s_x, d_C = s_z, t_z = d_B\}[AE]\{t_x = d_A, d_A = d_E, d_E = s_m + s_n\}[BCD]\{t_m = d_C, t_n = d_D, d_D = d_B, d_B = s_x, d_C = s_z, t_z = d_B\}[AE]\{t_x = d_A, d_A = d_E, d_E = s_m + s_n\}$

They are indexed as:
$R_1 = [BCD]\{t_m = c_{last-2}, t_n = d_{last-2}, d_{last-2} = b_{last-2}, b_{last-2} = s_x, c_{last-2} = s_z, t_z = b_{last-2}\}[AE]\{t_x = a_{last-2}, a_{last-2} = e_{last-2}, e_{last-2} = s_m + s_n\}[BCD]\{t_m = c_{last-1}, t_n = d_{last-1}, d_{last-1} = b_{last-1}, b_{last-1} = s_x, c_{last-1} = s_z, t_z = b_{last-1}\}[AE]\{t_x = a_{last-1}, a_{last-1} = e_{last-1}, e_{last-1} = s_m + s_n\}[BCD]\{t_m = c_{last}, t_n = d_{last}, d_{last} = b_{last}, b_{last} = s_x, c_{last} = s_z, t_z = b_{last}\}[AE]\{t_x = a_{last}, a_{last} = e_{last}, e_{last} = s_m + s_n\}$

By forming the transitive closure of the data constraints we will have the following (we only show the constraints which will be used in deriving the last expression):

$R_1 = [BCD]\{b_{last-2} = d_{last-2}, c_{last-2} = b_{last-3}, b_{last-2} = a_{last-3}\}[AE]\{a_{last-2} = e_{last-2}, e_{last-2} = d_{last-2} + c_{last-2}\}[BCD]\{d_{last-1} = b_{last-1}, b_{last-2} = c_{last-1}, b_{last-1} = a_{last-2}\}[AE]\{a_{last-1} = e_{last-1}, e_{last-1} = d_{last-1} + c_{last-1}\}[BCD]\{d_{last} = b_{last}, b_{last-1} = c_{last}, b_{last} = a_{last-1}\}[AE]\{a_{last} = e_{last}, e_{last} = d_{last} + c_{last}\}$

Here, we still have one more step to go. We shall move backward in the constraints and substitute the corresponding data elements: $e_{last} = d_{last} + c_{last}$, then $e_{last} = b_{last} + b_{last-1}$, then $e_{last} = a_{last-1} + a_{last-2}$, and in the end we have $e_{last} = e_{last-1} + e_{last-2}$.

Thus, finally this yields the following symbolic output for the Fibonacci circuit:

$$e_{last} = e_{last-1} + e_{last-2}$$