Science of Computer Programming 77 (2012) 848–869

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Symbolic execution of Reo circuits using constraint automata

Bahman Pourvatan[a,b], Marjan Sirjani[c,d,*], Hossein Hojjat[e], Farhad Arbab[b,f]

[a] Computer Engineering Department, Amirkabir University of Technology, Tehran, Iran
[b] LIACS, Leiden University, The Netherlands
[c] School of Computer Science, Reykjavik University, Reykjavik, Iceland
[d] ECE Department, University of Tehran, Tehran, Iran
[e] EPFL, Lausanne, Switzerland
[f] CWI, The Netherlands

A R T I C L E   I N F O

Article history:
Received 7 December 2009
Received in revised form 5 April 2011
Accepted 6 April 2011
Available online 6 May 2011

Keywords:
Symbolic execution
Reo
Constraint automata
Coordination languages
Program verification
Program validation

A B S T R A C T

Reo is a coordination language that can be used to model different systems. We propose a technique for symbolic execution of Reo circuits using the symbolic representation of data constraints in Constraint Automata. This technique enables us to obtain the relations among the data that pass through different nodes in a circuit from which we infer the coordination patterns of the circuit. Deadlocks, which may involve data values, can also be checked using reachability analysis. As an alternative to constructing the symbolic execution tree, we use regular expressions and their derivatives which we obtain from our deterministic finite automata. We show that there is an upper bound of *one* for unfolding the cycles in constraint automata which is enough to reveal the relations between symbolic representations of inputs and outputs. In the presence of feedback in a Reo circuit a number of substitutions are necessary to make the relationship between successive input/output values explicit. The number of these substitutions depends on the number of buffers in the Reo circuit, and can be found by static analysis. We illustrate our technique on a set of Reo circuits to show the extracted relations between inputs and outputs. We have implemented a tool to automate our proposed technique.

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 [2,3] were introduced as compositional semantics for Reo. Using constraint automata (CA) we can analyze the behavior of Reo circuits.

Reo has been used in different applications including composition of Web services [4–9], modeling and analysis of long-running transactions and compliance in service-oriented systems [10,11], hardware–software design [12,13], coordination of multi-agent systems [14], and modeling of coordination in biological systems [15]. Reo and constraint automata are used as an ADL (Architectural Description Language) in [16]. In most of these applications, Reo is generally used to represent the communication and synchronization, and constraint automata are used to model the components. In some cases the behavior of the whole system is compositionally constructed using the constraint automata of its constituents.

* Corresponding author at: School of Computer Science, Reykjavik University, Reykjavik, Iceland.
E-mail addresses: pourvatan@aut.ac.ir (B. Pourvatan), marjan@ru.is (M. Sirjani), hossein.hojjat@epfl.ch (H. Hojjat), farhad@cwi.nl (F. Arbab).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.04.001

Tool support for Reo consists of a set of Eclipse plug-ins that together comprise the Eclipse Coordination Tools (ECT) visual programming environment [17]. The Reo graphical editor supports drag-and-drop graphical composition and editing of Reo circuits. The animation plugin automatically generates a graphical animation of the flow of data in a Reo circuit, which provides an intuitive insight into their behavior through visualization of how they work. A converter plugin automatically generates the CA model of a Reo circuit.

In this paper, we present our novel work on symbolic execution of Reo using constraint automata. Our technique can be used to derive the symbolic output of a circuit in terms of its input data that represents the coordination pattern implemented by the circuit. It can also reveal possible deadlocks/livelocks. Several tools are available for analyzing Reo. One is the Vereofy model checker presented in [18,19], which is integrated in the ECT. Another is the symbolic model checker of the mCRL2 toolset [20,21]. An automated tool integrated in the ECT translates Reo models into mCRL2 and provides a bridge to its tool set. This translation tool and its application to the analysis of timed data-aware workflows modeled in Reo are discussed in [22,23].

Our symbolic execution technique and tool constitute a simple yet powerful analyzer which can be used as an alternative to model checking tools. Whereas in model checking, one can express more complex properties and pay a higher computational cost for their analysis, symbolic execution of Reo circuits can be used effectively for deriving their coordination patterns. Similar techniques can be used for slicing reductions and test case generation for Reo and CA.

The main idea behind symbolic execution [24] 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 their symbolic input values [25]. While symbolic execution tools and model checkers are both used for analyzing systems, symbolic execution is 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 typically 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 theory of automata. 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 a previously encountered state of a constraint automaton (or in some cases to other states which are considered as final or accepting states). Speaking in the context of Reo, this means that all enabled nodes fire and the circuit returns to a previous configuration (or a stable 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, treating any previously encountered state as the final state [26]. To do this, we define the set of regular expressions associated with the CA, considering how synchrony and asynchrony are both modeled in CA, and show how data constraints can be represented in regular expressions. Our approach relies on the manipulation of data constraints in different execution paths.

The main difficulties in performing symbolic execution are (1) dealing with a potentially infinite number of symbolic execution paths; (2) examining path feasibility through constraint solving or similar approaches; and (3) handling complicated data structures and data types. In this paper, we show that instead of dealing with an infinite number of execution paths, for constraint automata, it is enough to traverse each cycle only once, yielding a finite number of paths. If there is a feedback in a Reo circuit, we may have recursive relations between the data streams. In that case, we need a number of substitutions to uncover the relation. This number can be computed by traversing the cycle once. For the constraint solving problem and cutting the infeasible paths, we have the same difficulties as the other approaches [27,28,25]. However, it rarely happens that we experience complex conditions in our coordination patterns. For the third problem, there is some work on addressing the problem of error detection for programs that take recursive data structures and arrays as input [29]. We abstract from complicated data structures and data types as they are not regularly considered in Reo and constraint automata, focusing instead on deriving nontrivial interdependencies among data.

Contribution. We provide an analysis technique for Reo circuits based on the symbolic execution approach. Reo is used for different applications but the only tools provided for its analysis are a few model checking tools [30,31,18]. Our symbolic execution technique has the following highlights:

- The technique derives the symbolic output of a circuit in terms of its input data and yields its coordination patterns.
- The technique provides reachability analysis and reveals possible deadlocks/livelocks which are usually the most interesting properties for designers.
- This technique is based on symbolic representation of the variables and not their actual values. Therefore, it does not need the data domain to be finite. In contrast, model checking needs to consider the actual values of variables.
- This technique starts from automata models instead of program code. Therefore, it can avoid building the typical symbolic execution trees and instead generate and unfold regular expressions. The coordination patterns are then uncovered using derivatives of these regular expressions.
- To the best of our knowledge, this is the first work that uses data constraints in constraint automata to perform the analysis on Reo circuits. The regular expressions that we define for constraint automata incorporate their data constraints.
- We propose a solution for the potentially infinite number of symbolic executions according to the semantics of Reo circuits.
- We have developed an automated tool for symbolic execution of Reo which will be embedded in the ECT tool.

Plan of the paper. This paper is an extended version of Pourvatan et al. at FOCLASA 2009 [32]. For this extension, we added a thorough study of the literature to provide an overview and position our work among other similar work. We also present the formal underpinning of our approach. We expanded the discussion of the case studies to illustrate the applicability of our method.

The rest of this paper is organized as follows: Section 2 is a brief overview of the coordination language, Reo, and constraint automata. In Section 3, we review the general concept of symbolic execution. In Section 4, we explain our approach for symbolic execution of Reo circuits. Section 5 contains a number of case studies. Related work is presented in Section 6. We discuss the applicability and limits of our technique in Section 7. In Section 8, we discuss future work and conclude the paper. In Appendix A, we provide a brief overview of Brzozowski's method for deriving a regular expression from an automaton. In Appendix B, we provide the details of applying our approach on the case studies discussed in Section 5, and in Appendix C, we present our experimental results for these case studies.

2. Background: coordination languages, Reo, and constraint automata

In this section we review coordination languages, and more specifically Reo and its constraint automata semantics.

2.1. Coordination languages

The increasingly more complex software-intensive systems of today involve intricate interactions among multitudes of constituents (e.g., agents, threads, processes, objects, components, etc.) and exhibit the need for explicit attention to their *coordination*. Abstracting away from computation makes coordination activity different than and independent of specific applications. Coordination activity in all applications involves a common set of primitive concepts that center on interaction [33].

A broad survey [34] of coordination models and languages concluded that coordination models can be categorized as data-driven or control-driven. In data-driven models such as Linda [35] and its extensions, coordination tends to be endogenous and embedded within computational entities. The activity in a data-oriented application tends to center around a substantial shared body of data; the application is essentially concerned with what happens to the data. Examples include database and transaction systems like in banking and airline reservation applications. The activity in control-oriented application tends to center around processing or flow of control and, often, the very notion of *the data* simply does not exist. Applications that involve work-flow in organizations are examples of this type. Control-driven models such as ROAD [36], IWIM [37], and CoLaS [38] isolate coordination by considering functional entities as black boxes. In control-driven models, coordination can be exogenous and isolated from the coordinated computational entities. For instance, IWIM addresses computation and coordination concerns in separate and independent levels. Hybrid approaches such as tuple center [39] and ReSpecT [40,41] combine the data-driven and control-driven models.

In a new classification presented in [33], a third group is added to this classification as dataflow-oriented models. These models use the flow of data as the only (or at least the primary) control mechanism. Unlike data-oriented models, dataflow models are primarily oblivious to the actual content, type, or structure of data and are instead concerned with the flow of data from their sources to their destinations. Unlike control-oriented models, events that trigger state transitions are limited to only those that arise out of the flow of data. Reo is classified as a dataflow-oriented coordination language.

The symbolic execution technique proposed in this paper is based on the constraint automata semantics of Reo [3]. As such, it may also be applicable to other coordination languages whose semantics can be expressed as constraint automata. Below, we show this potential by illustrating the work on the common semantic foundation and the mappings among a number of coordination models and constraint automata. The details of possible necessary customization and/or limits of our technique for other languages are out of scope of this paper.

In [42,43] the authors compare Reo and two actor-based coordination languages, Actors–Roles–Coordinators (ARC) [44] and Policy-based Reflective Russian Dolls (PBRD) [45] and explain a first step towards a common semantic foundation for the three models. Whereas actor languages [46] are generally classified as control-oriented, ARC and PBRD can both be classified as dataflow-oriented languages. In both models messages that can be considered as flow of data cause the state transitions. These messages are not sent from actor to actor, but are manipulated on their way by the coordinators. In [42,43] it is shown that the semantics of interaction in all models can be mapped into a common representation that can be captured by constraint automata. In an earlier work, a compositional semantics of an actor-based language is defined using constraint automata [47].

In a different line of work, the orchestration language BPEL [48], and the choreography language WS-CDL [49] are mapped into Reo and constraint automata [7,8]. These mappings cover a large subset of BPEL and WS-CDL languages. The set of Reo channels that are used in these mappings is the same as the set we use in this paper. As such, our symbolic execution technique is in principle applicable to the outcome of these mappings.

2.2. Reo

Reo is a model for building component connectors in a compositional manner [1]. It allows us to model the behavior of such connectors, formally reason about them, and once proven correct, automatically generate the so-called glue code from these models. 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 enter and *sink end* through which data leave 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].

A set of primitive Reo channels (together with their constraint automata that will be explained later in this section) are shown in Fig. 1. Channels are connected to make a circuit. Connecting (or *joining*) channels is putting channel ends together in *nodes*. Thus, a set of channel ends is associated with a *node*. 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 channel ends coincident 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. Fig. 2(b) shows a Reo circuit (together with its constraint automaton) that results from a join of two simple channels (*FIFO1* channels). In Fig. 2(b) node A is a source node, node C is a sink node, and node B is a mixed node. Node B consists of two channel ends, one channel end of type sink from the *FIFO1* on the left and one channel end of type source from the *FIFO1* on the right.

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 an open ended set of channels, but a set of primitive channels, shown in Fig. 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 (blocking) I/O operations through that connector, which itself is oblivious of those entities. This makes Reo a powerful *glue language* for compositional construction of connectors to combine component instances and Web services into a software system and exogenously orchestrate their mutual interactions.

2.3. Constraint automata: compositional semantics of Reo

Constraint automata are presented in [3,2] as a formal semantics for Reo connectors, based on a co-algebraic semantics given in [50]. 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 in a Reo connector) while the automata-transitions represent the possible data flows and their effects 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, operators are defined to compose the constraint automata of the primitives used in a Reo connector.

A constraint automaton is defined in [2] as follows:

Definition 1 (*Constraint Automaton*)**.** A constraint automaton (over the data domain *Data*) is a tuple $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ where

- Q is a set of states,
- $\mathcal{N}$ is a finite set of names,
- $\longrightarrow$ is a subset of $Q \times 2^{\mathcal{N}} \times DC \times Q$, 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 is a data constraint (defined below). $\mathcal{A}$ is called finite iff Q, $\longrightarrow$ and the underlying data domain *Data* are finite.
Data constraints consist of $g \in DC(N, Data)$, where $DC(N, Data)$ is the language defined by the following grammar [2]:

$$g ::= \mathsf{true} \mid \neg g \mid u = u \mid g \wedge g$$
$$u ::= d_n \mid v.$$

Here, n denotes a name in N, d_n denotes the data item exchanged through node n, and $v \in Data$ denotes a data item. We assume a global data domain *Data* for the data exchanged through all names. Alternatively, we can assign a data domain $Data_A$ to every name $A \in N$ 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 $d_A = d_B$ which is a short-hand for $d_A = d \wedge d_B = d$ for all $d \in Data$, or $d_A \ni P$ which is a short-hand for $d_A = d$ for all $d \in P \subseteq Data$.

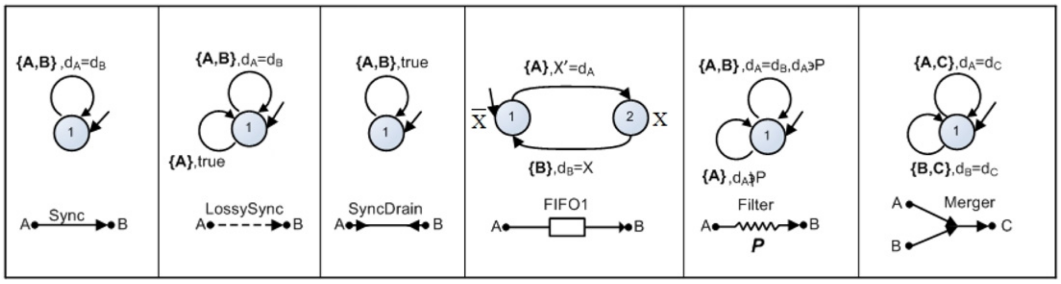


Fig. 1. Reo building blocks and their constraint automata.

Satisfiability and validity, logical equivalence $\equiv$, and logical implication $\implies$ 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 the constraint automaton for each of the basic connectors and the automata-operation product to mimic the behavior of the Reo-operation join.

Fig. 1 shows the constraint automata for the merger node and for the standard basic channel types. A *Sync* 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 *SyncDrain* channel has 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. Fig. 1.c shows a *LossySync* channel is similar to the *Sync* channel, except that 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 behaves like the *Sync* channel except that it loses all data that do not match the specified pattern of the filter (P in the figure). The *FIFO1* channel has a source (A) and a sink end (B), and a bounded buffer with capacity of 1 data item (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 chooses the input from one of the channel ends A or B nondeterministically, and passes it to the channel end C as its output. A product operator is defined on constraint automata that captures the meaning of Reo's join operator [2]. The product-automaton of the two constraint automata $\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \longrightarrow_1, Q_{0,1})$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \longrightarrow_2, Q_{0,2})$ is given by:

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

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

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

and

$$\frac{q_1 \overset{N_1,g_1}{\longrightarrow}_1 p_1, \quad N_1 \cap \mathcal{N}_2 = \emptyset}{\langle q_1, q_2 \rangle \overset{N_1,g_1}{\longrightarrow} \langle p_1, q_2 \rangle}$$

and the latter's symmetric rule (see [3,2]). The first rule is applied when there are two transitions that can fire together. This happens only if the two transitions agree on their shared names. Node names occurring in both automata are either present in both transitions or in neither of them. In this case the resulting transition in the product automaton has the union of the names on both transitions, and its 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 fire independently of the other automaton, which happens when the names on the transition are not included in the other automaton.

An extension of constraint automata with State Memory (CASM), explicitly partitions $\mathcal{N}$, into three disjoint sets of $\mathcal{N}^{src}$, $\mathcal{N}^{snk}$, and $\mathcal{N}^{mix}$ and extends data constraints to accommodate state memory cells [51]. Here, we use a slightly simplified form of the CASM, using the state memories and leaving out the partitioning of the name set.

Definition 2 (*Constraint Automaton with State Memory (CASM)*)**.** A constraint automaton with state memory (over the data domain *Data*) is a tuple $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ where [51]:

- Q is a finite set of states
- $\mathcal{N}$ is a finite set of names.
- $\rightarrow$ is a finite subset of $Q \times 2^{\mathcal{N}} \times DC(\mathcal{N}, \mathcal{M}, Data) \times Q$, called the transition relation of $\mathcal{A}$, where $DC(\mathcal{N}, \mathcal{M}, Data)$ is the set of data constraints, defined below.
- $q_0 \in Q$ is an initial state.
- $\mathcal{M}$ is a set of memory cell names, where $\mathcal{N} \cap \mathcal{M} = \emptyset$.

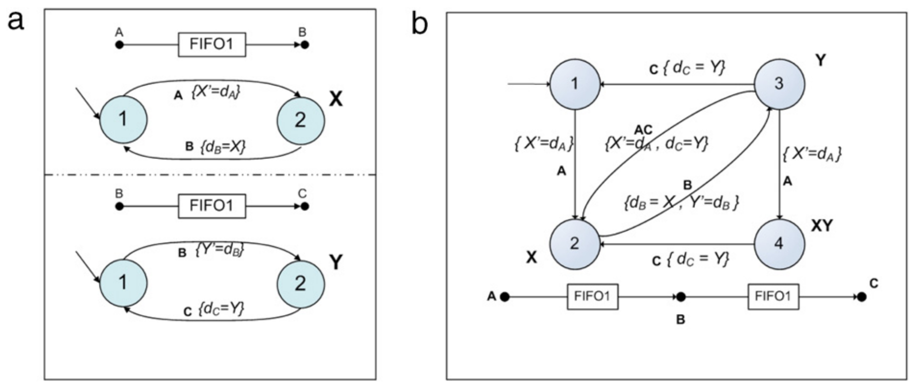


Fig. 2. Constraint automata with state memory for (a) *FIFO1* and (b) *FIFO2*.

We can partition $\mathcal{N}$ into three disjoint sets, $\mathcal{N}^{src}$ a set of source node names, $\mathcal{N}^{snk}$ a set of sink node names, and $\mathcal{N}^{mix}$ a set of mixed node names. Thus $\mathcal{N} = \mathcal{N}^{src} \uplus \mathcal{N}^{snk} \uplus \mathcal{N}^{mix}$.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \rightarrow$. We call $N \in 2^{\mathcal{N}}$ the name-set and g the guard or data constraint of the transition. For every transition $q \xrightarrow{N,g} p$ we require that $g \in DC(N, \mathcal{M}, Data)$, where $DC(N, \mathcal{M}, Data)$ is the language defined by the following grammar:

$$g ::= true \mid \neg g \mid g \wedge g \mid u = u$$
$$u ::= d_n \mid m' \mid m \mid v.$$

Here "=" is the symmetric equality relation, $n \in N$ is a port name, d_n refers to the data item that passes through port n, $m \in \mathcal{M}$ refers to a memory cell in the current state (source of the transition), m' refers to the memory cell m in the next state (target of the transition), and $v \in Data$. As a shorthand, we allow in our syntax *false* to stand for $\neg$ *true*, and other logical operators, such as $\vee$ and $\Longrightarrow$ (for implication), can also be defined, in the usual way.

We omit transitions whose data constraints can be reduced to $\neg$ *true* using the Boolean laws. A data constraint, g, that can be reduced to *true* can be left out. We use $\mathcal{M}_g$ to denote the set of all $m \in \mathcal{M}$ that syntactically appear as m in a data constraint g; and $\mathcal{M}'_g$ to denote the set of all $m \in \mathcal{M}$ that syntactically appear as m' in g.

We use a valuation function $\mathcal{V}_q : \mathcal{M} \to 2^{Data}$ to designate the set of values $\mathcal{V}_q(m)$ of a memory cell $m \in \mathcal{M}$ in a state $q \in Q$, where $\mathcal{V}_{q_0}(m) = \emptyset$ for all $m \in \mathcal{M}$. A constraint automaton with state memory can make a transition $q \xrightarrow{N,g} p$ only if there exists a substitution for every syntactic element d_n, m, and m' that appears in g to make it *true*. A substitution simultaneously replaces every occurrence of d_n with the data value (to be) exchanged through the node $n \in N$; every occurrence of m with a value $v \in \mathcal{V}_q(m)$; and every occurrence of m' with a value $v \in Data$. Making this transition, the automaton defines the valuation function $\mathcal{V}_p$ for the target state p, as follows. For every $m \in \mathcal{M}'_g$, $\mathcal{V}_p(m)$ is the set of all $v \in Data$ whose replacements in g yield substitutions that make g *true*. For every $m \in \mathcal{M} \setminus \mathcal{M}'_g$, $\mathcal{V}_p(m) = \emptyset$.

The product of two CASM is defined similarly to the product of two constraint automata where the set of memory cells of the product is the union of the sets of memory cells of its two operands. The rules that define the set of transitions of the product are the same as in the product of constraint automata.

We can see the CASM for two *FIFO1* channels in Fig. 2(a). In this figure X and Y are the internal buffers of the two *FIFO1* channels. Joining the 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* (a FIFO with buffer capacity of 2) channel as in Fig. 2(b). The product of the CASM of the two channels gives us the CASM of a *FIFO2* channel. In this example, we see how the data constraints on the values in 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, are related. In the rest of this paper, for simplicity, we use constraint automata or its abbreviation, CA, instead of CASM.

3. Background: symbolic execution

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 [24,52,53]. 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 [28].

We use a simple example to illustrate the technique [54]. The actual execution of a program requires concrete values (e.g., $x = 3, y = 10$) as its input data. In contrast, symbolic execution uses symbols as values of variables. For instance, $x = a_0, y = b_0$. The execution usually involves operations on complex expressions. Suppose there is a statement like $z = x + 2y$. With ordinary program execution, the result of z will be a concrete value like 23. But with symbolic execution, the result is typically some symbolic expression like $a_0 + 2b_0$. After many steps of execution, a variable's value may become a very complex expression. A key problem with symbolic execution is to decide whether a set of constraints is satisfiable. Suppose there is a statement *if* $(z < x)$ $S1$ *else* $S2$. Then we need to know whether $x + 2y < x$ holds in the current context. If there is a precondition saying that y must be positive, the condition will not be satisfiable. Thus the statement $S2$ will be executed and $S1$ will not. Otherwise, we may need to execute both $S1$ and $S2$ and build both paths.

There is some trade-off between the expressiveness of a constraint language and the difficulty of deciding the satisfiability of its constraints [55]. For instance, the satisfiability of the conjunctive quantifier-free formulas in linear arithmetic constraints over rational numbers is decidable in polynomial time. The satisfiability of linear arithmetic constraints over integers and natural numbers is NP-complete. Non-linear arithmetic over integers is undecidable [56].

4. Symbolic execution of Reo circuits

For symbolic execution of Reo circuits we use constraint automata (CA) as their semantics. In CA, instead of specific data values, we already have a symbolic representation of input and output values. Moreover, the data constraints on the transitions show the possible relations among these symbolically represented data values. This makes CA an appropriate basis to build a symbolic execution tool. Subsequently, we use the data constraints to derive the relation between input and output values.

We need to obtain a symbolic representation of the possible relations between output and input value. We use a path-based analysis in symbolic execution. A constraint automaton can be treated as a directed graph, wherein a path is defined as usual for graphs. To enumerate the possible execution paths, we generate the symbolic execution tree of a given Reo circuit using its constraint automaton. Alternatively, we can obtain the regular expression of a CA and obtain the execution paths as the derivatives of these regular expressions.

In this paper we use a set of primitive Reo channels that consists of *Sync*, *SyncDrain*, *LossySync*, *FIFO1*, and *Filter* (shown in Fig. 1). In [1] it is shown by construction that this set captures regular expressions. However, by using these primitives, we have only equalities among the data elements that pass through the ports. Also, the primitive channels and consequently their products involve only conjunction as Boolean connective (although complex constraints are allowed for filter channels; see Section 7). It follows that these primitives generate deterministic CA with single initial states.

4.1. 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 corresponding edge in the traversal tree.

As we construct the tree, every time we see a name (port or node in Reo) it means that a new data element is observed in the stream of data passing through that port [50]. For a node name A, we use $\tilde{A}$ to denote the (finite) data stream that passes through A, as:

$$\tilde{A} = (a_0, a_1, \ldots, a_{n-1}, a_n) \quad \text{where for } 0 \le i \le n,\, a_i \in Data.$$

To write the data constraints for each state in the traversal tree, we use the last elements of these (finite) data streams $\tilde{A} = (a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}, a_n)$. For the set of all finite streams *FStreams* over a set of elements *Element*, we define *last* : *FStreams* $\times \mathbb{N}^- \cup \{0, 1\} \longrightarrow$ *Element*, where $\mathbb{N}^-$ is the set of negative integers, as a function such that $last(\tilde{A}, i)$ takes a finite stream $\tilde{A}$ and an integer $i \le 0$ and returns the ith last element of $\tilde{A}$. Thus, $last(\tilde{A}, 0)$ is the last element of the stream $\tilde{A}$ and $last(\tilde{A}, -1)$ is the first next to last element of $\tilde{A}$, etc. Using the function *last* on streams, we can rewrite a stream using negative indices, going backward from its last element, as: $\tilde{A} = (last(\tilde{A}, -n), last(\tilde{A}, -(n-1)), last(\tilde{A}, -(n-2)), \ldots, last(\tilde{A}, -2), last(\tilde{A}, -1), last(\tilde{A}, 0))$. For convenience, we use a superscript notation to refer to the function *last* and place its second argument as a subscript index, writing a^{ℓ}_i instead of $last(\tilde{A}, i)$. Thus, we write the stream $\tilde{A}$ as $\tilde{A} = (a^{\ell}_{-n}, a^{\ell}_{-(n-1)}, a^{\ell}_{-(n-2)}, \ldots, a^{\ell}_{-2}, a^{\ell}_{-1}, a^{\ell})$, where we drop the subscript zero on the last term a^{ℓ}_0. We treat the sequence of

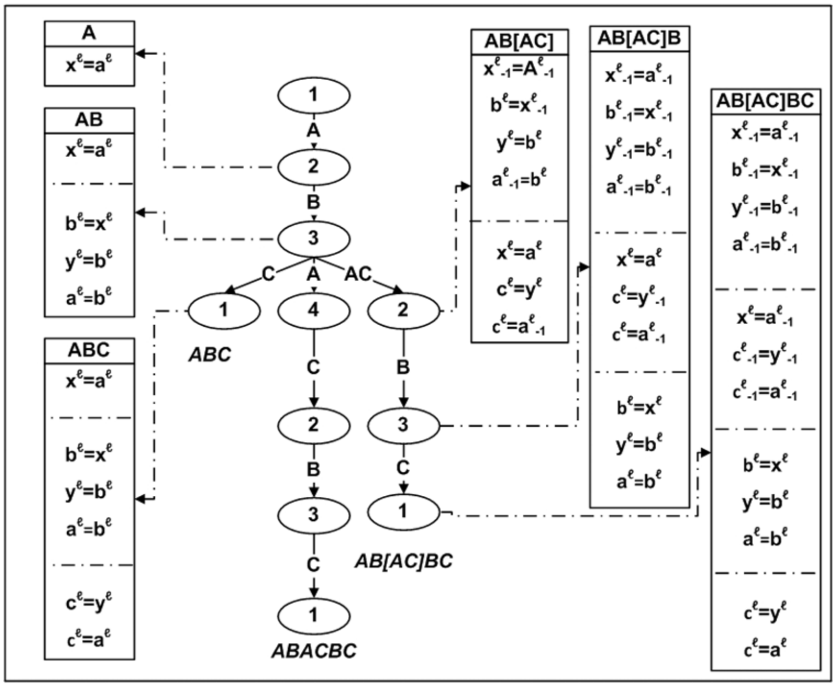


Fig. 3. Symbolic execution tree for *FIFO2*: each node represents a state and edges represent transitions in the CA. The string of a leaf lists the ports that fire consecutively to reach that leaf from the root; a pair of square brackets in this string encloses a set of port names that fire synchronously. Note that this is not a complete execution tree.

values assigned to each memory cell as a finite stream and backward-index it accordingly (there are special cases where for a memory cell we refer to its value in the future using the subscript of $+1$).

Fig. 3 shows the execution tree for a *FIFO2* channel. In this figure the relations among data elements are shown. The boxes connected to each node of the tree show the names that fire to reach that tree node (CA state) together with all the data constraints that hold at that point (have been met until that point), plus the data constraints that can be derived from them. Consider the boxes on the left-hand side of Fig. 3. The upper box shows that A fires and $x^\ell = a^\ell$. The lower box shows that A and then B fire and the set of data constraints consists of $x^\ell = a^\ell$ that is generated in the previous transition (firing of A), and $b^\ell = x^\ell$ and $y^\ell = b^\ell$ that are added by the firing of B, plus the data constraint $a^\ell = b^\ell$ which is inferred from other constraints.

We generate all relevant execution paths from the initial state back to a previously visited state. The string on each leaf of the tree shows the names of the ports that fire along its corresponding path. A pair of square brackets encloses a set of port names that fire synchronously, like A and C that fire synchronously in a transition from state 3 to state 2 in Fig. 2(b), shown as $[AC]$ in the symbolic execution tree in Fig. 3.

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). The set of primitive Reo channels introduced above, yield data constraints involving only the equality relation. Hence, the relations corresponding to the data passing through the nodes are symmetric. The transitive closure of this relation gives us all symbolic relations among input and output values (that pass through the ports of the circuit).

The symbolic execution tree in Fig. 3 is generated by traversing each cycle in the CA only once. In this example, this is sufficient to yield the general expression for the relation between input and output values. No new relation will be generated by traversing the cycles in this example 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 the input data. In this case, we focus only on the paths that include the name of this particular output node.

4.2. Coordination patterns and unfolding of cycles

A constraint automaton $\mathcal{A}$ imposes a (generally infinite) relation among the data elements that pass through its nodes and/or are stored in its memory cells. What is interesting about this relation is not so much the specific data values that it inter-relates, but the relative position of each such value in its respective stream. Thus, we define a relation $R_{\mathcal{A}} \subseteq \mathcal{T} \times \mathbb{N} \times \mathcal{T} \times \mathbb{N}$ where $\mathcal{T} = \mathcal{M} \cup \mathcal{N}$, such that $(X, i, Y, j) \in \mathcal{R}_{\mathcal{A}}$ means that $\mathcal{A}$ relates the ith element of $\tilde{X}$ (the stream of values exchanged through X) with the jth element of $\tilde{Y}$ (the stream of values exchanged through Y). The exact meaning of "relates" depends on the nature of the constraints that appear in the data constraints on the transitions of $\mathcal{A}$. In our case, data constraints involve only equality, which makes $\mathcal{R}_{\mathcal{A}}$ symmetric. Intuitively, the transitive closure of this relation, $\mathcal{R}^*_{\mathcal{A}}$, expresses everything that $\mathcal{A}$ does as a coordinator. Thus, we refer to any finite formal representation of (a subset of) $\mathcal{R}^*_{\mathcal{A}}$ as the (or, a) *coordination pattern* of $\mathcal{A}$.

Generally, both the number and the lengths of the execution paths in the symbolic execution tree of a Reo circuit can become infinite, because of the existence of cycles in its CA, which means $\mathcal{R}_{\mathcal{A}}$ and $\mathcal{R}^*_{\mathcal{A}}$ are infinite. However, the infinite tuples in $\mathcal{R}_{\mathcal{A}}$ arise out of a finite pattern of inter-dependencies that can be discovered by traversing every cycle only once.

Consider a cycle C in a CA $\mathcal{A}$ that arises out of visiting states q_0 to q_n and back to q_0, through transitions $q_i \xrightarrow{N_i,g_i} q_{i+1}$ for $0 \leq i < n$, and then $q_n \xrightarrow{N_n,g_n} q_0$. The cycle C consists of $n+1$ transitions. The first time that $\mathcal{A}$ makes the kth transition in C, the appearance of the names A and B in a relation in the data constraint g_k of this transition relates the data element a_i of the stream $\tilde{A}$ with the data element b_j of the stream $\tilde{B}$. Thus g_k establishes $(A, i, B, j) \in \mathcal{R}_{\mathcal{A}}$. The data constraints of the n other transitions in C contain $x \geq 0$ references to A, a number which we call the *depth index* of A in cycle C, denoted by A^d_C. Thus, $A^d_C = x$, and similarly, the depth index of B in C is $B^d_C = y$ for some $y \geq 0$. Repeating the cycle, the second time that $\mathcal{A}$ encounters the kth transition in C, its data constraint g_k relates the data element a_{i+x} of the stream $\tilde{A}$ with the data element b_{j+y} of the stream $\tilde{B}$. This establishes $(A, i + A^d_C, B, j + B^d_C) \in \mathcal{R}_{\mathcal{A}}$. Going the third time through C to reach the kth transition yields $(A, i + 2 \times A^d_C, B, j + 2 \times B^d_C) \in \mathcal{R}_{\mathcal{A}}$. Generalizing, if $\mathcal{A}$ goes through the cycle c times, we get $(A, i + c \times A^d_C, B, j + c \times B^d_C) \in \mathcal{R}_{\mathcal{A}}$ from the kth transition in C.

We treat every state that $\mathcal{A}$ reaches at the end of every cycle as if it were a final state; i.e., we assume that $\mathcal{A}$ may stop, at which point all streams have their final lengths. As such, the cycle C, thus, relates in $\mathcal{R}_{\mathcal{A}}$ the last A^d_C elements of the finite stream $\tilde{A}$ with some of the last elements of other finite streams, including the last B^d_C elements of the finite stream $\tilde{B}$. Using the negative backward indexing scheme of Section 4.1, $\tilde{A} = (\ldots, a_i, a_{i+1}, a_{i+2}, \ldots, a_{i+A^d_C-2}, a_{i+A^d_C-1}, a_{i+A^d_C})$ can be changed to $\tilde{A} = (\ldots, a^\ell_{-A^d_C}, a^\ell_{1-A^d_C}, a^\ell_{2-A^d_C}, \ldots, a^\ell_{-2}, a^\ell_{-1}, a^\ell)$. This allows us to get rid of the "base indices" such as i and j in the tuples of the relation $\mathcal{R}_{\mathcal{A}}$ by, for instance, replacing the tuples $(A, i + A^d_C, \_, \_), \ldots, (A, i, \_, \_)$ with $(A, 0, \_, \_), \ldots, (A, -A^d_C, \_, \_)$.

Observe that by traversing a cycle C only once, we obtain the depth index X^d_C of every (node or memory cell) name X that appears in every transition in C. Given these indices for all names in all cycles of $\mathcal{A}$, we can generate the entire $\mathcal{R}_{\mathcal{A}}$. Any compact expression of a relation of interest among a set of (node or memory cell) names of $\mathcal{A}$ implied by $\mathcal{R}^*_A$ represents a coordination pattern of $\mathcal{A}$.

4.3. Regular expressions and path feasibility

In the setting of verification of (typically imperative) programs, symbolic execution techniques naturally build their respective symbolic execution trees. But in our case, we have the advantage of the availability of the constraint automata instead of the source code of a program. Thus, instead of building the symbolic execution tree, we can use the regular expression of the CA to obtain more compact representations of their execution paths.

We generate the regular expression of a CA using Brzozowski's algebraic method [26,57,58] (see Appendix A for more details). This method is based on knowing the final state of the automaton. A final state is not defined in CA. To apply Brzozowski's method, we treat any state of the automaton that it reaches at the end of a cycle, and any state that has no outgoing transition, as a final state. Each state in a CA represents the configuration of its respective Reo circuit. Our assumption reflects the intuition that the CA is in a final state when it is back in a state corresponding to a previously visited configuration of its Reo circuit. We obtain the execution paths of a Reo circuit as the derivatives of the regular expressions of its CA. By unfolding the data constraints, as explained in Section 4.2, along these paths we obtain the coordination patterns of the CA (and the Reo circuit).

As we produce the regular expressions of a CA, we carry the data constraints specified on the transitions. The expression that we derive from the regular expression includes the condition which shows the feasibility of that derivative, or rather the condition under which the coordination pattern holds. All data constraints in the set of primitive channels that we consider are equalities. Therefore, the complexity of path feasibility analysis depends on the complexity of filter patterns used in the Reo circuit. In our current tool implementation, we accommodate only equality constraints as filter patterns.

4.4. Main algorithm

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

(1) Obtain the regular expression for the constraint automaton of the circuit.
(2) Construct the derivatives (words) of the regular expression.
(3) Build the symbolic data stream of each output node based on the symbolic input stream of data for each derivative.

In the following, we explain each step and use the *FIFO2* circuit shown in Fig. 2(b) as a running example.

Step 1: *Produce the regular expression.* We use Brzozowski's algebraic method [26] and Arden's theorem [59] for generating regular expressions for constraint automata. We create a system of regular expressions with one unknown regular expression for each state in a constraint automaton, and then we solve the system for R_{q_0} where R_{q_0} is the regular expression associated with the initial state q_0. 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 nodes 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 in square brackets), 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 expression for the *FIFO1* channel in Fig. 2(a), whose initial state is 1, is: $R_1 = (AB)^*$. Augmenting it with data constraints we have $R_1 = (A\{X' = d_A\}B\{d_B = X\})^*$. For our running example, the *FIFO2* of Fig. 2(b), we have
$R_1 = ((A)(B[AC] + BAC)^*BC)^*$
and the complete expression including the data constraints is:
$R_1 = ((A\{X' = d_A\})(B\{d_B = X, Y' = d_B\}[AC]\{X' = d_A, d_C = Y\} +$
$B\{d_B = X, Y' = d_B\}A\{X' = d_A\}C\{d_C = Y\})^*B\{d_B = X, Y' = d_B\}C\{d_C = Y\})^*$
For *LossySync*, we add the data constraint $\{d_A = d_A\}$ on the transition with the name A to indicate the loss of data.

Step 2: *Produce all derivatives for the regular expression by substituting each '$*$' with zero or one repetitions.* In this step, we generate all possible derivatives of the regular expression by considering the repetition in the expression to be zero or one (see Section 4.2). This means that we traverse a cycle only once. 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 once (substitution of '$*$' with one), or bypass the cycle where possible (substitution of '$*$' with zero).

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

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

- Substituting the inner '$*$' with zero repetition and the outer '$*$' with one:
  $R_{1_2} = (ABC)$
  and including data constraints:
  $R_{1_2} = (A\{X' = d_A\}B\{d_B = X, Y' = d_B\}C\{d_C = Y\}).$

- Substituting the inner '$*$' with one repetition and the outer '$*$' with one:
  $R_{1_3} = (A(B[AC] + BAC)BC)$
  and including data constraints:
  $R_{1_3} = (\,A\{X' = d_A\}(B\{d_B = X, Y' = d_B\}[AC]\{X' = d_A, d_C = Y\} + B\{d_B = X, Y' = d_B\}A\{X' = d_A\}C\{d_C = Y\})$
  $B\{d_B = X, Y' = d_B\}C\{d_C = Y\}).$

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 reveal its data streams by indexing their elements and the references to the memory cells. We traverse every sequential term of the regular expression from right to left, and specify indices for each data element and memory cell in the data constraints occurring in the regular expression. Then, all transitive relations among these elements of streams and memory cells (transitive closure) give us the relation between output and input values.

Step 3.1: *Backward Indexing.* Intuitively, the purpose of indices is to show the order of the nodes seen along the traversal of a regular expression. We traverse every sequential term of the regular expression from right to left. For example, if we observe a name A for the first time in a regular expression (the rightmost occurrence), we replace its data d_A with the indexed name a^ℓ. If we see another A we denote the new one as a^ℓ_{-1}, and so on.

Indexing of variables needs more care. The typical scenario is that when we start from the right and move to the left, for each memory cell X, we first see X and we replace it with x^ℓ. Then, we see X' and we replace it with x^ℓ too. Note that we are moving backward in our indexing process, and observing X' (writing to memory cell X) shows that we are writing an element into the stream $\tilde{X}$, and observing X (reading from the memory cell X) shows that we are reading from the stream $\tilde{X}$ (i.e., we read the data that was previously written to the memory cell X). The rules for substituting the indices of memory references are the same as for references to node names, except that if the first encounter (from right to left) with a memory cell name X is of the syntactic form X', we replace it with x^ℓ_{+1}. Such a reference writes a value into the memory cell X of the "final state" leaving it there as the "initial value" for the next cycle in the run of the automaton.

For the *FIFO2* example, we obtain the following equation for the second derivative in step 2:
$R_{1_2} = (A\{x^\ell = a^\ell\}B\{b^\ell = x^\ell, y^\ell = b^\ell\}C\{c^\ell = y^\ell\}).$

Note that in indexing the data elements, the starting index for all subexpressions of a '+' is the same. The equation for the last regular expression in step 2 is:
$R_{1_3} = (A\{x^{\ell}_{-1} = a^{\ell}_{-1}\}$
$(B\{b^{\ell}_{-1} = x^{\ell}_{-1}, y^{\ell}_{-1} = b^{\ell}_{-1}\}[AC]\{x^{\ell} = a^{\ell}, c^{\ell}_{-1} = y^{\ell}_{-1}\} + B\{b^{\ell}_{-1} = x^{\ell}_{-1}, y^{\ell}_{-1} = b^{\ell}_{-1}\}A\{x^{\ell} = a^{\ell}\}C\{c^{\ell}_{-1} = y^{\ell}_{-1}\})$
$B\{b^{\ell} = x^{\ell}, y^{\ell} = b^{\ell}\}C\{c^{\ell} = y^{\ell}\})$.

Double indexing. When we finish traversing a parenthesized expression for indexing, we may end up with two different indexed versions of a 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 $R = (A\{X' = d_A\}([AB]\{d_B = d_A\} + B\{d_B = X\}))$ we compute the indices for A (and B) as follows:
$R = (A\{(x^{\ell} = a^{\ell}) \vee (x^{\ell} = a^{\ell}_{-1})\}([AB]\{b^{\ell} = a^{\ell}\} + B\{b^{\ell} = x^{\ell}\}))$.

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

$$R_{1_2} = (A\{x^{\ell} = a^{\ell}\}B\{b^{\ell} = x^{\ell}, y^{\ell} = b^{\ell}\}C\{c^{\ell} = y^{\ell}\}).$$

We can write:

$$R_{1_2} = (AB\{x^{\ell} = a^{\ell}, b^{\ell} = x^{\ell}, y^{\ell} = b^{\ell}, a^{\ell} = b^{\ell}, y^{\ell} = a^{\ell}, x^{\ell} = y^{\ell}\}C\{c^{\ell} = y^{\ell}\}).$$

And then:
$R_{1_2} = (ABC\{x^{\ell} = a^{\ell}, b^{\ell} = x^{\ell}, y^{\ell} = b^{\ell}, a^{\ell} = b^{\ell}, y^{\ell} = a^{\ell}, x^{\ell} = y^{\ell}, c^{\ell} = y^{\ell}, c^{\ell} = a^{\ell}, c^{\ell} = b^{\ell}, c^{\ell} = x^{\ell}\})$.

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:

$$R_{1_2} = (ABC\{c^{\ell} = a^{\ell}\}).$$

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 it would imply atomic firings.)

We continue for R_{1_3}:
$R_{1_3} = ((A\{x^{\ell}_{-1} = a^{\ell}_{-1}\})$
$(B\{b^{\ell}_{-1} = x^{\ell}_{-1}, y^{\ell}_{-1} = b^{\ell}_{-1}\}[AC]\{x^{\ell} = a^{\ell}, c^{\ell}_{-1} = y^{\ell}_{-1}\} + B\{b^{\ell}_{-1} = x^{\ell}_{-1}, y^{\ell}_{-1} = b^{\ell}_{-1}\}A\{x^{\ell} = a^{\ell}\}C\{c^{\ell}_{-1} = y^{\ell}_{-1}\})$
$B\{b^{\ell} = x^{\ell}, y^{\ell} = b^{\ell}\}C\{c^{\ell} = y^{\ell}\})$.

Then:

$$R_{1_3} = (A(B[AC] + BAC)BC\{a^{\ell}_{-1} = c^{\ell}_{-1}, c^{\ell} = a^{\ell}\}).$$

This gives us the relations: $c^{\ell} = a^{\ell}, c^{\ell}_{-1} = a^{\ell}_{-1}$, 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 its corresponding element of a (with the same index) in the input node of the Reo circuit.

4.5. Iterating the cycle in the presence of a feedback

If we have a feedback in a Reo circuit we may be interested in finding the recursive relation between the data items passing through a node. Consider for example the *FIFO2* channel of Fig. 2(b), if we put a *Sync* channel from the node C back to the node A then we have feedback (for the circuit to have at least one enabled node to fire we need to have one data item in one of the buffers). In this case, we still can obtain the relations between the output C and the input A of the *FIFO2* by traversing each cycle of its CA only once. However, the existence of a feedback loop implies a relationship between the successive values that pass through the same node, e.g., A in our example. If we are interested in this relation, then we need to know the relation between a^{ℓ} and a^{ℓ}_{-i} (for some i). In the case of *FIFO2* we have $a^{\ell} = a^{\ell}_{-1}$.

To distinguish a feedback in a Reo circuit from its corresponding CA, we look at all the tuples in the relation we derived by traversing a cycle (see Section 4.2). If we observe a memory cell or a node name with two different indices in the relation it means that there may be a feedback. Intuitively, this represents a data item getting back into a buffer where it has already been. For example, from (a^{ℓ}, x^{ℓ}) and $(a^{\ell}, x^{\ell}_{-2})$, we can derive $(a^{\ell}, a^{\ell}_{-2})$ by substitution. The upper bound for the number of substitutions is the number of memory cells in the relation built for a cycle.

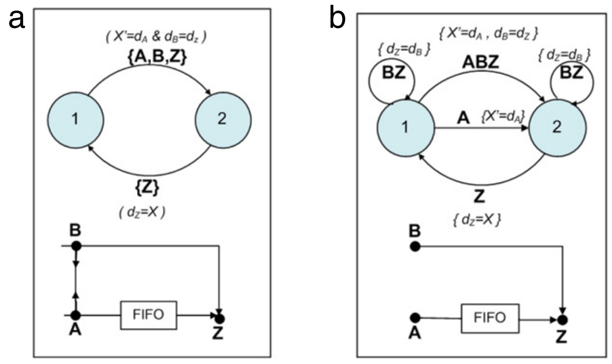


Fig. 4. (a) Alternator and (b) Nondeterministic choice, and their constraint automata.

4.6. *Finding deadlocks and livelocks*

We can use symbolic execution to find possible deadlocks or livelocks. To do this, we apply the same technique to derive the relation between inputs and outputs. Starting with a specification, its designer expects a relation between the data stream on an output Reo node, say O, and a set of input data streams entering a set of Reo nodes I. We apply our technique trying to find such relations. If no relation is found, it means that the data streams that come in from the Reo nodes in the set I, never reach the Reo node O (based on the discussions in Section 4.2). This shows an error in the Reo circuit which may be a livelock or a deadlock.

In [32] we showed an example of a system with two processes and a critical section. 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 can occur. Thus, we detected a deadlock in the system without generating the whole state space, which is hence less expensive than traditional model checking.

5. Case studies

In this section we present a few case studies as examples. For all Reo circuits we assume that data is always available (offered) at their input ports and acceptable at their output ports. In the following examples, we derive the relation between the output data on the output ports and the inputs of the circuit. These relations cannot immediately be inferred from the regular expressions of the CA of the circuit without applying our symbolic execution analysis using data constraints.

Our tool is implemented in the Eclipse environment. The inputs of the tool are the CA and the names of the input and output ports whose inter-relations we are interested in. The output of the tool is the relations among inputs and outputs as shown in the following case studies. For each derivative (cycle), we do the indexing, derive the transitive closure, and give the relation between inputs and outputs. The time complexity of extracting all derivatives is exponential in terms of number of states. The space complexity of this process is $n + m^2$ where n is the number of states in each path corresponding to a cycle, and m is the number of names in the CA. In our algorithm, we process only one derivative at a time, so the algorithm is efficient regarding memory usage. For a CA with 2×10^6 transitions, our tool uses 700MB of heap.

Example 1 (*Alternator Circuit*)**.** The alternator circuit is shown in Fig. 4(a). In this circuit, data enter from ports A and B, and exit from port Z. Data enter ports B and A synchronously because of the *SyncDrain* channel BA. The data item from port A is buffered in the *FIFO1* channel (which is empty in the initial state) and the data item from port B exits from port Z immediately (because of the *Sync* Channel BZ). The data stream observed on port Z is $(ba)^*$ [1]. 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 Appendix B. The final result is:

$$z^{\ell}_{-1} = b^{\ell}, \qquad z^{\ell} = a^{\ell}.$$

This shows the pattern of the output: (1) the output data from port Z are formed by alternating between the input data from ports B and A, (2) the data from B is synchronously produced on Z, and (3) the data from A is produced on Z with a one cycle step.

Example 2 (*Nondeterministic Choice*)**.** Fig. 4(b) shows a Reo circuit with a nondeterministic choice between its input ports. This circuit is similar to the **Alternator** circuit in Example 1 except that the *SyncDrain BA* is removed. The *FIFO1* channel is empty in the initial state. Because of the nondeterministic choice in the *merger* at node Z, the output data exiting the port Z can be from input port A with one cycle step or from input port B synchronously. 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 Appendix B. The final result is:

$$(z^{\ell}_{-1} = b^{\ell}, z^{\ell} = a^{\ell}) \;\vee\; (z^{\ell} = b^{\ell}) \;\vee\; (z^{\ell} = a^{\ell}).$$

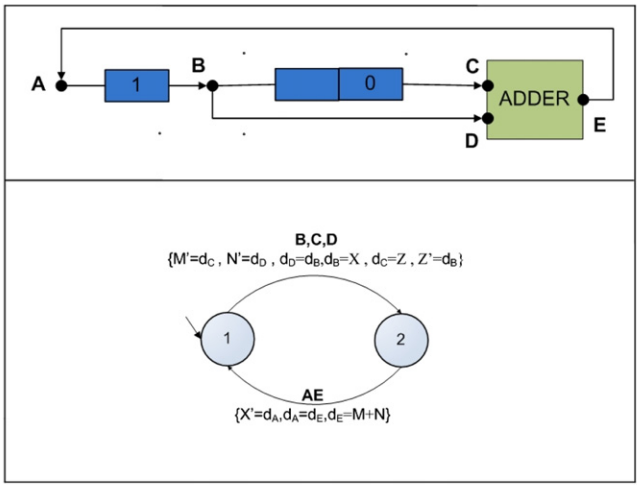


Fig. 5. Fibonacci Reo circuit and its constraint automaton.

This shows all three possible relations between the input and the output values.

Example 3 (*Fibonacci Series*)**.** The Reo circuit for the Fibonacci series [3] has a feedback from its output to its input to generate values of this recursive formula. Fig. 5 shows the 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 E feeding back as an input to the same circuit at A. To extract the recursive relation, we need to use substitution as explained in Section 4.5. In this example, X is the memory cell for *FIFO1*, Y and Z are the memory cells for *FIFO2*, and M and N are the memory cells corresponding to the inputs of the Adder component. The Adder component repeatedly gets its two input values into its memory cells M and N, adds them, and puts the result 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 Appendix B.

To derive the relation between the sequence of data on node E of the Reo circuit with itself, 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 we traverse the cycle only once, and do substitutions 3 times). Thus, finally this yields the following symbolic output for the Fibonacci circuit:

$$e^{\ell} = e^{\ell}_{-1} + e^{\ell}_{-2}.$$

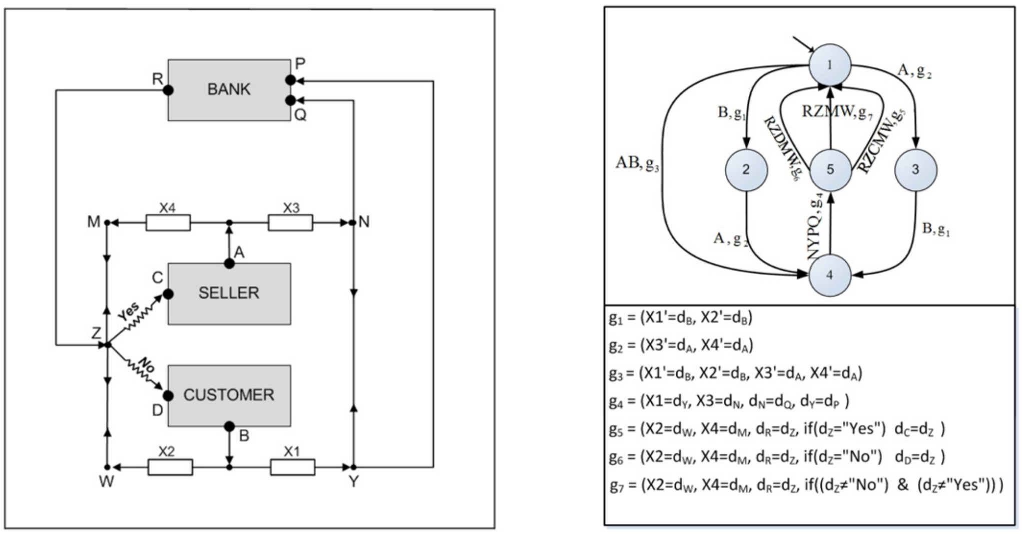
Example 4 (*Banking Transaction*)**.** We now show an example that involves coordinating a banking transaction. We have three components: a seller, a customer, and a bank. The seller sends its account number and product price to the bank, and the customer sends its account number and pay order to the bank. The bank checks the balance of the customer's account. If the bank accepts the transaction, it sends a `Yes` to the seller, and otherwise a `No` to the customer. The Reo circuit and its CA for coordinating this transaction are shown in Fig. 6. In the CA we can see that the information can be sent by the *Seller* and *Customer* in any order or simultaneously, and the *Bank* replies after receiving the information from both. The CA for a *Filter* channel is shown in Fig. 1. In Fig. 6, the *Filter ZC* passes only the message *Yes* to the *Seller* component and the *Filter ZD* passes only the message *No* to the *Customer* component.

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 Appendix B. The final result is:

$if\,(r^{\ell} = \boldsymbol{Yes})\ c^{\ell} = \boldsymbol{Yes} \quad \vee \quad if\,(r^{\ell} = \boldsymbol{No})\ \ d^{\ell} = \boldsymbol{No}.$

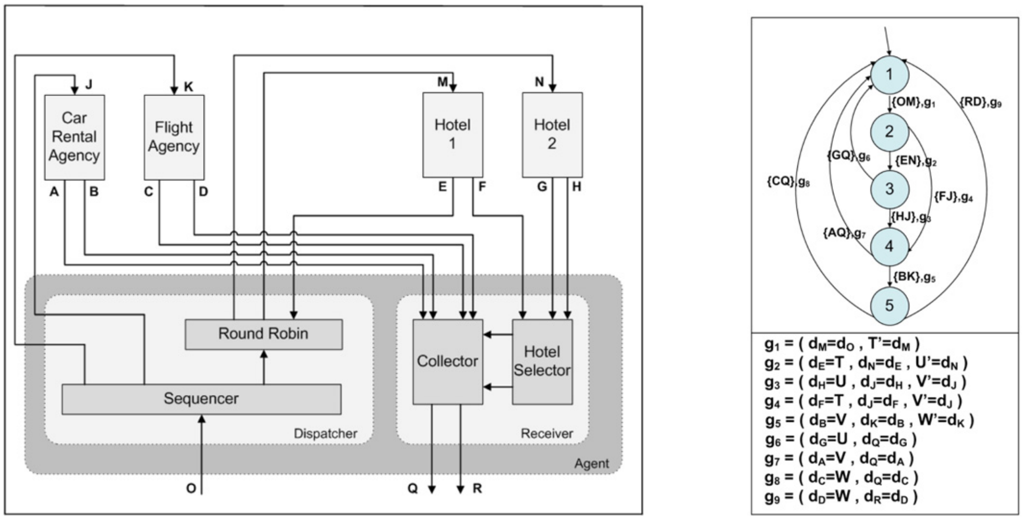
Example 5 (*Travel Agency*)**.** As a larger case study, we consider a travel agency that coordinates the process of reserving a hotel, issuing the flight ticket, and renting a car for its customers. Fig. 7 shows the Reo circuit and the CA for the agent and three components: the car rental agency, the ticket service, and two hotels. We can extend this example to have more car rental agencies, ticket services and hotels. The number of states for the CA will be $(M + N + P + 1)$ where M is the number of car rental agencies, N the number of ticket services, and P the number of hotels. The number of transitions in this case will be $(2 \times (M + N + P) + 1)$ and the number of cycles will be $(M + N + P + 3)$. When we have more than one car rental agency or ticket service, then as for the hotels, we need a selector component in the agent.

In Fig. 7, O is the request port, R is the port for the positive response, and Q is the port for the negative response from the agent. The names J, K, M and N refer to the input ports of the components, A, C, E and G are the ports for the negative responses, and B, D, F and H are the ports for the positive responses. Based on the relation between the output of the travel agency and the inputs from the other components we observe that the successful output becomes true only if the inputs from the flight ticket and car rental component and one of the hotels are true (see Appendix B).



$g_1 = (X1'=d_B, X2'=d_B)$
$g_2 = (X3'=d_A, X4'=d_A)$
$g_3 = (X1'=d_B, X2'=d_B, X3'=d_A, X4'=d_A)$
$g_4 = (X1=d_Y, X3=d_N, d_N=d_Q, d_Y=d_P)$
$g_5 = (X2=d_W, X4=d_M, d_R=d_Z, \text{if}(d_Z=\text{"Yes"})\ d_C=d_Z)$
$g_6 = (X2=d_W, X4=d_M, d_R=d_Z, \text{if}(d_Z=\text{"No"})\ d_D=d_Z)$
$g_7 = (X2=d_W, X4=d_M, d_R=d_Z, \text{if}((d_Z\neq\text{"No"})\ \&\ (d_Z\neq\text{"Yes"})))$

Fig. 6. Banking transaction Reo circuit and its constraint automaton.



$g_1 = (d_M=d_O, T'=d_M)$
$g_2 = (d_E=T, d_N=d_E, U'=d_N)$
$g_3 = (d_H=U, d_J=d_H, V'=d_J)$
$g_4 = (d_F=T, d_J=d_F, V'=d_J)$
$g_5 = (d_B=V, d_K=d_B, W'=d_K)$
$g_6 = (d_G=U, d_Q=d_G)$
$g_7 = (d_A=V, d_Q=d_A)$
$g_8 = (d_C=W, d_Q=d_C)$
$g_9 = (d_D=W, d_R=d_D)$

Fig. 7. Travel Agency Reo circuit and its constraint automaton.

6. Related work

We look into related work from different perspectives. First, we discuss symbolic execution versus model checking to compare our work with the other existing tools for analyzing Reo circuits. Then we discuss related works that apply symbolic execution on hardware and software in general, or combine symbolic execution with other analysis techniques like testing. Finally, we discuss a few recent works that use recurrent patterns to deal with a potentially infinite number of symbolic execution paths.

Symbolic execution versus model checking

Model checking, like symbolic execution, is an analysis technique to check the behavior of a system. Model checking is a formal verification method whereas symbolic execution is a program analysis technique. A formal verification method typically consists of three major components: a model for describing the behavior of the system, a specification language to express correctness requirements, and an analysis technique to verify the behavior against the correctness requirements [60,61]. Given a model and a correctness property, an automated model checking tool returns either a counter example to show that the property does not hold, or a positive answer which means that the property is satisfied. The property specification languages used usually consist of variants of temporal logic, like LTL (Linear Temporal Logic) and CTL (Computational Tree Logic) [62].

Model checking techniques are based on generating the state space of the model and checking whether or not the specified properties are violated. State space explosion is the main obstacle in model checking, as to generate the state space, every occurrence of a data variable is expanded to every possible value that the variable can assume. Different abstraction and reduction techniques are proposed to tackle this problem. For instance, one way to tackle state space explosion is to manipulate a set of states (rather than a single state) at a time, where a set of states is represented by a formula in propositional logic. This approach is called symbolic model checking [63,64] and is used, e.g., in implementing the CTL model checking tool for Reo [18]. The term "symbolic" here is unrelated to the term "symbolic" in symbolic execution.

Model checking can be used to verify liveness and safety properties of a system. For instance, when two trains moving in opposite directions must pass through the same one-way bridge, both trains eventually passing through the bridge is a liveness property, and the two trains never being on the bridge at the same time is a safety property.

Symbolic execution, in its traditional form, is used to construct a symbolic representation of the outputs from the inputs, and to explore different feasible paths in a program. Symbolic execution is classified as a static analysis technique where the subject program is neither executed on concrete data values, nor is its state space generated. More recently, this technique has been used in various ways in combination with other analysis techniques, for instance, model checking [25] and testing [27,65].

Symbolic execution of hardware and software

Symbolic execution is an established technique in verification, especially in the hardware verification community. Having its roots in the seventies [24,52,53], symbolic execution rapidly grew to be one of the predominant verification techniques for hardware designs, with many successful applications, such as [66]. With the invention of symbolic trajectory evaluation by Seger and Bryant in the nineties [67] this approach has now become a component in most of the industrial hardware verification systems (e.g., FORTE [68] at Intel). As Reo has previously been used for modeling hardware systems [12,13], 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 symbolically represent 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 an analysis technique for software systems as well. The degree of nondeterminism and the large number of different factors that determine the result of a software system usually curtail the application of symbolic execution as the only analysis technique in practice, without any other accompanying technique. Examples include [69] 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. The similarity of this work and our work is in the attempt 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 [25] 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 [28]. The main concern in this work is handling complicated data structures and loops effectively. In [65], 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 above works constitute a set of active and ongoing research on symbolic execution with a focus on resolving the problems with details of implementation and complicated data structures.

In [70] a method for verifying the correctness of parallel programs is presented. The authors of this work check the equivalence of a parallel program and its sequential version which serves as the specification for the parallel version. 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.

Recurrent patterns

In the following works the authors tackle the state space explosion in ways other than data abstraction or bounded execution paths. The main idea is explained in [27]: "a reactive system is supposed to continuously interact with its environment. Thus, behaviors viewed as sequences of interactions are very often arbitrarily long. Specifications of reactive systems often contain internal loops and their symbolic executions have infinite paths. However, in practice, one can consider an arbitrarily long behavior as a sequence of basic behaviors. Reactive systems generally have the property that they regularly come back to already encountered states, as for example the initial state. For such systems, we can look for techniques to yield a finite tree by symbolic execution."

In [27], the authors propose an approach to test whether a system conforms to its specification. They use symbolic execution techniques to extract test purposes from the specification and to generate test cases for the system. Test purposes are expressed as symbolic execution paths of the specification. They introduced two criteria. The first criterion is called *all symbolic behaviors of length n*. The second criterion is called *restriction by inclusion*, which means that the extracted subtree satisfies a coverage criterion which is based on a procedure of redundancy detection. In other words, information on symbolic behaviors provided by the symbolic execution may be highly redundant in terms of basic behaviors. This approach detects the basic behaviors that are included in already computed symbolic behaviors. The restriction by inclusion criterion extracts a subset of all paths of the symbolic execution tree by avoiding redundancies.

The authors of [29] use state matching techniques to limit the state space instead of their previous attempts where they put a bound on the size of the program inputs and/or the search depth of the model checker. They work on the problem of error detection for programs with recursive data structures and arrays as input. They proposed a method for examining whether a symbolic state that arises during symbolic execution is subsumed by another symbolic state. The number of symbolic states may be infinite and so, subsumption may not be enough to ensure termination. Therefore, they consider abstraction techniques for computing and storing abstract states during symbolic execution. Subsumption checking determines whether an abstract state is being revisited, in which case the model checker backtracks. This enables the analysis of an under-approximation of the program behaviors.

The "idea" in these works is similar to ours in finding the redundant patterns of interactions. The difference is that we have the advantage of starting from CA as our specification, and can use a simple technique to find the sufficient bound for symbolic execution.

7. Discussion

In this section we discuss the applicability and the limits of our technique. First, we discuss the three main problems of symbolic execution: constraint solving, data structures, and infinite execution paths. Reo is proposed as a coordination language. The main purpose of Reo circuits is to act as glue code among the collaborating components, facilitating their communication and synchronization. The components are responsible for performing complex computations. Of course, in a general setting, according to the definition of Reo, filter channels can have all kinds of predicates as their accepting patterns. This will add to the complexity of our constraints. However, in dealing with typical coordination Reo circuits, we do not encounter undecidable constraint solving problems in our symbolic execution approach. We carry the constraints while executing the paths to the end. In the case that the constraints become more complicated, we will face the same problems as in constraint solving.

A similar argument applies for complex data structures. Although complex data structures may be transferred between components, the coordinator usually does not deal with all the details of such data items; the decision for communication or synchronization is usually based on simple data. The third problem concerns the loops and cycles that produce infinite execution paths. We proposed our simple solution based on the semantics of Reo and constraint automata.

We can apply symbolic execution on other types of automata, but as shown in this paper, our technique performs well because of the specific characteristics of constraint automata: (1) we have synchronization constraints, (2) data constraints on each transition are strictly local and can refer to only those ports named in the synchronization constraints and the source and target states of that transition, and (3) we have symbolic representation for data streams. For example, in I/O Automata [71] considering the form of the constraints that are collected during the traversal of the automaton and the fact that a modified variable in each transition can refer to any variable in the state of the whole system, there may not be a general rule to decide the sufficient number of iterations through a loop. This suggests that finding recurring patterns and a bound on generating the execution paths is not necessarily possible in the case of I/O automata. In general, the applicability of our technique for other kinds of automata is not trivial and needs further investigation.

CA are dataflow-oriented models that support synchrony and asynchrony at the same time. Several languages are mapped into Reo and CA [12,7,8,72,73], and as mentioned in Section 2.1, in principle our approach can be applied to these languages as well. However, the effectiveness of the approach depends on the complexity of the predicates in the conditions and the complexity of potential data structures used.

8. 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 a previously encountered state, passing each cycle only once. 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 future work, we consider applying different methods for improving our technique and tool. We are studying the compositionality of the technique where we can use the coordination patterns of two Reo circuits to derive the coordination patterns of their composition instead of deriving it from their product CA. We will study if and how we can substitute these expressions for each port when we compose two Reo circuits. This will significantly help in scalability of our technique. We will improve our tool implementation by using more efficient algorithms and data structures to be able to handle larger models.

We will investigate different applications amenable to our symbolic technique. More specifically, we consider looking into applying our technique on other automata.

Acknowledgements

We thank Carolyn Talcott for her insightful and very useful comments. We also thank the reviewers of the workshop version of our paper presented at FOCLASA'09, for their very careful and helpful comments. The extension of the paper is mainly based on their comments. They also pointed to the related work and literature which helped us to properly position our method among other approaches. The work of the second author is partly supported by CWI (within the NWO project "Synthesis and Analysis of Connector Components (SYANCO)" DN62-613).

Appendix A. Brzozowski's algebraic method

We adapt Brzozowski's method [58,26] to generate regular expressions from constraint automata. For a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, q_0, \mathcal{M})$, we create a system of regular expressions with one unknown regular expression R_{q_i} for every state in $q_i \in Q$, and then we solve the system for R_{q_0}, the regular expression associated with the initial state q_0. These equations are the characteristic equations of Q. Constructing the characteristic equations is straightforward. For each state $q_i \in Q$, the equation for R_{q_i} is a union of terms. Each term can be constructed as follows: for a transition from q_i to q_j which has a label a, the term aR_{q_j} is added. This leads to a system of equations of the form:

$$\begin{aligned}
&R_{q_0} = a_0R_{q_0} + a_1R_{q_1} + a_2R_{q_2} + \cdots\\
&R_{q_1} = a_0R_{q_0} + a_1R_{q_1} + a_2R_{q_2} + \cdots\\
&R_{q_2} = a_0R_{q_0} + a_1R_{q_1} + a_2R_{q_2} + \cdots\\
&R_{q_3} = a_0R_{q_0} + a_1R_{q_1} + a_2R_{q_2} + \cdots\\
&\cdots\cdots\\
&R_{q_m} = a_0R_{q_0} + a_1R_{q_1} + a_2R_{q_2} + \cdots
\end{aligned}$$

where the term $a_iR_{q_j}$ appears on the right-hand-side of the equation defining R_{q_i} if and only if there is a transition from q_i to q_j with a_i as its label. If q_0 is an accepting state, ϵ is added to R_{q_0} as one of the terms.

$$R_{q_0} = a_0R_{q_0} + a_1R_{q_1} + a_2R_{q_2} + \cdots + \epsilon.$$

By equation substitution, we get $R_{q_0} = AR_{q_0} + B$, where A and B are expressions of transition labels for an accepting state R_{q_0}. Arden's theorem [59] can resolve these situations. This theorem states that these equations have the solution $R_{q_0} = A^*B$ where the language of A does not contain ϵ.

In our setting, where we use constraint automata, a transition label consists of a set of node names and guards. For example, consider the constraint automaton for *FIFO1* in Fig. 1. Its characteristic equations are as follows (where the initial state is 1 and the other state is 2):

$R_1 = A\{X' = d_A\}R_2 + \epsilon$
$R_2 = B\{X = d_B\}R_1$.
We substitute into R_1 and solve:
$R_1 = A\{X' = d_A\}B\{X = d_B\}R_1 + \epsilon$.
We solve for R_1 using Arden's theorem:
$R_1 = (A\{X' = d_A\}B\{X = d_B\})^*\epsilon$ or
$R_1 = (A\{X' = d_A\}B\{X = d_B\})^*$.

Appendix B. 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:
$R_1 = ([ABZ]Z)^*$
including data constraints:
$R_1 = ([ABZ]\{X' = d_A, d_B = d_Z\}Z\{d_Z = X\})^*$.
The regular expression derivative:
$R_{1_1} = ([ABZ]Z)$
including data constraints:
$R_{1_1} = ([ABZ]\{X' = d_A, d_B = d_Z\}Z\{d_Z = X\})$.
It is indexed as:
$R_{1_1} = ([ABZ]\{x^\ell = a^\ell, b^\ell = z^\ell_{-1}\}Z\{z^\ell = x^\ell\})$.
Forming the transitive closure of the data constraints, we have:
$R_{1_1} = ([ABZ]Z\{x^\ell = a^\ell, b^\ell = z^\ell_{-1}, x^\ell = z^\ell, z^\ell = a^\ell\})$.
And finally we have:

$$z^\ell_{-1} = b^\ell, \qquad z^\ell = a^\ell.$$

Example 2: Nondeterministic 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:
$R_1 = ([BZ] + ([ABZ] + A)([BZ])^*Z)^*$
including data constraints:
$R_1 = ([BZ]\{d_Z = d_B\} + ([ABZ]\{X' = d_A, d_B = d_Z\} + A\{X' = d_A\})([BZ]\{d_Z = d_B\})^*Z\{d_Z = X\})^*$.
The regular expression derivatives:
$R_{1_1} = ([BZ] + ([ABZ] + A)([BZ])Z)$
$R_{1_2} = ([BZ] + ([ABZ] + A)Z)$
including data constraints:
$R_{1_1} = ([BZ]\{d_Z = d_B\} + ([ABZ]\{X' = d_A, d_B = d_Z\} + A\{X' = d_A\})([BZ]\{d_Z = d_B\})Z\{d_Z = X\})$
$R_{1_2} = ([BZ]\{d_Z = d_B\} + ([ABZ]\{X' = d_A, d_B = d_Z\} + A\{X' = d_A\})Z\{d_Z = X\})$.
They are indexed as:
$R_{1_1} = ([BZ]\{z^\ell = b^\ell\} + ([ABZ]\{x^\ell = a^\ell, b^\ell_{-2} = z^\ell_{-2}\} + A\{x^\ell = a^\ell\})([BZ]\{z^\ell_{-1} = b^\ell\})Z\{z^\ell = x^\ell\})$
$R_{1_2} = ([BZ]\{z^\ell = b^\ell\} + ([ABZ]\{x^\ell = a^\ell, b^\ell = z^\ell_{-1}\} + A\{x^\ell = a^\ell\})Z\{z^\ell = x^\ell\})$.
Forming the transitive closure of the data constraints, we have:
$R_{1_1} = ([BZ]\{z^\ell = b^\ell\} + ([ABZ]\{x^\ell = a^\ell, b^\ell_{-1} = z^\ell_{-2}\} + A\{x^\ell = a^\ell\})([BZ]\{z^\ell_{-1} = b^\ell\})Z\{z^\ell = x^\ell\})$
$R_{1_2} = ([BZ]\{z^\ell = b^\ell\} + ([ABZ]\{x^\ell = a^\ell, b^\ell = z^\ell_{-1}\} + A\{x^\ell = a^\ell\})Z\{z^\ell = x^\ell\})$.
And finally we have:

$$z^\ell_{-1} = b^\ell, \qquad z^\ell = a^\ell \vee z^\ell = b^\ell \vee z^\ell = a^\ell.$$

Example 3: Fibonacci series

In this example, because of the feedback, we are interested in deriving the relation between the sequence of data that pass through node E in the Reo circuit with itself. Therefore, we need substitution in order to have a closed expression to show the interdependency of the data values in this sequence, i.e., to have a data item that passes through E specified in terms of earlier data items that have passed through E. 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 follows:

The regular expression:
$R_1 = ([BCD][AE])^*$
including data constraints:
$R_1 = ([BCD]\{M' = d_C, N' = d_D, d_D = d_B, d_B = X, d_C = Z, Z' = d_B\}[AE]\{X' = d_A, d_A = d_E, d_E = M + N\})^*$.
The regular expression derivative:
$R_{1_1} = [BCD][AE]$
including data constraints:
$R_{1_1} = [BCD]\{M' = d_C, N' = d_D, d_D = d_B, d_B = X, d_C = Z, Z' = d_B\}[AE]\{X' = d_A, d_A = d_E, d_E = M + N\}$.

We repeat the substitution until we obtain an expression that specifies the sequence of data on E in terms of itself. In this example, we need to repeat this substitution process three times, which is equal to the total number of buffers in the Reo circuit. Thus, we have:

$R_{1_1} = [BCD]\{M' = d_C, N' = d_D, d_D = d_B, d_B = X, d_C = Z, Z' = d_B\}$
$[AE]\{X' = d_A, d_A = d_E, d_E = M + N\}$
$[BCD]\{M' = d_C, N' = d_D, d_D = d_B, d_B = X, d_C = Z, Z' = d_B\}$
$[AE]\{X' = d_A, d_A = d_E, d_E = M + N\}$
$[BCD]\{M' = d_C, N' = d_D, d_D = d_B, d_B = X, d_C = Z, Z' = d_B\}$
$[AE]\{X' = d_A, d_A = d_E, d_E = M + N\}$.
They are indexed as:
$R_{1_1} = [BCD]\{m^\ell_{-2} = c^\ell_{-2}, n^\ell_{-2} = d^\ell_{-2}, d^\ell_{-2} = b^\ell_{-2}, b^\ell_{-2} = x^\ell_{-2}, c^\ell_{-2} = z^\ell_{-2}, z^\ell_{-1} = b^\ell_{-2}\}$
$[AE]\{x^\ell_{-1} = a^\ell_{-2}, a^\ell_{-2} = e^\ell_{-2}, e^\ell_{-2} = m^\ell_{-2} + n^\ell_{-2}\}$
$[BCD]\{m^\ell_{-1} = c^\ell_{-1}, n^\ell_{-1} = d^\ell_{-1}, d^\ell_{-1} = b^\ell_{-1}, b^\ell_{-1} = x^\ell_{-1}, c^\ell_{-1} = z^\ell_{-1}, z^\ell = b^\ell_{-1}\}$
$[AE]\{x^\ell = a^\ell_{-1}, a^\ell_{-1} = e^\ell_{-1}, e^\ell_{-1} = m^\ell_{-1} + n^\ell_{-1}\}$
$[BCD]\{m^\ell = c^\ell, n^\ell = d^\ell, d^\ell = b^\ell, b^\ell = x^\ell, c^\ell = z^\ell, z^\ell_{+1} = b^\ell\}$
$[AE]\{x^\ell_{+1} = a^\ell, a^\ell = e^\ell, e^\ell = m^\ell + n^\ell\}$.
By forming the transitive closure of the data constraints we obtain the following (we show only the constraints that will be used in deriving the final expression):
$R_{1_1} = [BCD]\{b^\ell_{-2} = d^\ell_{-2}, c^\ell_{-2} = b^\ell_{-3}, b^\ell_{-2} = a^\ell_{-3}\}$
$[AE]\{a^\ell_{-2} = e^\ell_{-2}, e^\ell_{-2} = d^\ell_{-2} + c^\ell_{-2}\}$
$[BCD]\{d^\ell_{-1} = b^\ell_{-1}, b^\ell_{-2} = c^\ell_{-1}, b^\ell_{-1} = a^\ell_{-2}\}$
$[AE]\{a^\ell_{-1} = e^\ell_{-1}, e^\ell_{-1} = d^\ell_{-1} + c^\ell_{-1}\}$

$[BCD]\{d^\ell = b^\ell, b^\ell_{-1} = c^\ell, b^\ell = a^\ell_{-1}\}$
$[AE]\{a^\ell = e^\ell, e^\ell = d^\ell + c^\ell\}$.

We still have one more step to go. Moving backward in the constraints and substituting the corresponding data elements, we get: $e^\ell = d^\ell + c^\ell$, then $e^\ell = b^\ell + b^\ell_{-1}$, then $e^\ell = a^\ell_{-1} + a^\ell_{-2}$, and finally, we obtain $e^\ell = e^\ell_{-1} + e^\ell_{-2}$.

Example 4: Banking transaction.

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 is:

$$R_1 = ((AB + BA + [AB])([NYPQ]([RZMW] + [RZMDW] + [RZCMW])))^*.$$

Intuitively, the first parenthesis $(AB + BA + [AB])$ takes the automaton from the initial state 1 to state 4 (via states 3 or 2 or directly). Then $[NYPQ]$ takes the automaton to state 5 and to get back to the initial state 1 (where possibly a new iteration can start) we can have $[RZMW]$ or $[RZMDW]$ or $[RZCMW]$. Augmenting the above expression with data constraint, we have:

$R_1 = ((A\{X3' = d_A, X4' = d_A\}B\{X1' = d_B, X2' = d_B\} +$
$B\{X1' = d_B, X2' = d_B\}A\{X3' = d_A, X4' = d_A\} +$
$[AB]\{X1' = d_B, X2' = d_B, X3' = d_A, X4' = d_A\})$
$([NYPQ]\{X1 = d_Y, X3 = d_N, d_N = d_Q, d_Y = d_P\}$
$([RZMW]\{X2 = d_W, X4 = d_M, d_R = d_Z, \mathit{if}((d_Z \neq Yes) \text{ and } (d_Z \neq No))\} +$
$[RZMDW]\{X2 = d_W, X4 = d_M, d_R = d_Z, \mathit{if}(d_Z = Yes) d_C = d_Z\} +$
$[RZMCW]\{X2 = d_W, X4 = d_M, d_R = d_Z, \mathit{if}(d_Z = No) d_D = d_Z\})))^*$.
They are indexed as:
$R_1 = ((A\{X3^\ell = a^\ell, X4^\ell = a^\ell\}B\{X1^\ell = b^\ell, X2^\ell = b^\ell\} +$
$B\{X1^\ell = b^\ell, X2^\ell = b^\ell\}A\{X3^\ell = a^\ell, X4^\ell = a^\ell\} +$
$[AB]\{X1^\ell = b^\ell, X2^\ell = b^\ell, X3^\ell = a^\ell, X4^\ell = a^\ell\})$
$([NYPQ]\{X1^\ell = y^\ell, X3^\ell = n^\ell, n^\ell = q^\ell, y^\ell = p^\ell\}$
$([RZMW]\{X2^\ell = w^\ell, X4^\ell = m^\ell, r^\ell = z^\ell, \mathit{if}((z^\ell \neq Yes) \text{ and } (z^\ell \neq No))\} +$
$[RZMDW]\{X2^\ell = w^\ell, X4^\ell = m^\ell, r^\ell = z^\ell, \mathit{if}(z^\ell = Yes) c^\ell = z^\ell\} +$
$[RZMCW]\{X2^\ell = w^\ell, X4^\ell = m^\ell, r^\ell = z^\ell, \mathit{if}(z^\ell = No) d^\ell = z^\ell\})))^*$.
And finally we have:
$\mathit{if}(r^\ell = \textbf{Yes}) c^\ell = \textbf{Yes} \quad \vee \quad \mathit{if}(r^\ell = \textbf{No}) d^\ell = \textbf{No}$.

Example 5: Travel agency.

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:

$R_1 = [OM]R_2 + \epsilon$
$R_2 = [EN]R_3 + [FJ]R_4$
$R_3 = [HJ]R_4 + [GQ]R_1$
$R_4 = [BK]R_5 + [AQ]R_1$
$R_5 = [RD]R_1 + [CQ]R_1 = ([RD] + [CQ])R_1$.

We solve the system of equations for R_1:
$R_1 = ([OM]([EN]([HJ]([BK]([RD] + [CQ]) + [AQ]) + [GQ]) + [FJ]([BK]([RD] + [CQ]) + [AQ])))^*$

$R_1 = ([OM][EN][HJ][BK][RD] + [OM][EN][HJ][BK][CQ] + [OM][EN][HJ][AQ] + [OM][EN][GQ] + [OM][FJ][BK][RD] + [OM][FJ][BK][CQ] + [OM][FJ][AQ])^*$.

Augmenting it with data constraint, we have:
$R_1 = ([OM]\{d_M = d_O, T' = d_M\}[EN]\{d_E = T, U' = d_N, d_E = d_N\}[HJ]\{d_H = U, V' = d_J, d_J = d_H\}[BK]\{d_B = V, W' = d_K, d_B = d_K\}[RD]\{d_D = W, d_R = d_D\} + [OM]\{d_M = d_O, T' = d_M\}[EN]\{d_E = T, U' = d_N, d_E = d_N\}[HJ]\{d_H = U, V' = d_J, d_J = d_H\}[BK]\{d_B = V, W' = d_K, d_B = d_K\}[CQ]\{d_C = W, d_Q = d_C\} + [OM]\{d_M = d_O, T' = d_M\}[EN]\{d_E = T, U' = d_N, d_E = d_N\}[HJ]\{d_H = U, V' = d_J, d_J = d_H\}[AQ]\{d_A = V, d_Q = d_A\} + [OM]\{d_M = d_O, T' = d_M\}[EN]\{d_E = T, U' = d_N, d_E = d_N\}[GQ]\{d_G = U, d_Q = d_G\} + [OM]\{d_M = d_O, T' = d_M\}[FJ][BK]\{d_B = V, W' = d_K, d_B = d_K\}[RD]\{d_D = W, d_R = d_D\} + [OM]\{d_M = d_O, T' = d_M\}[FJ]\{d_F = T, d_J = d_F, V' = d_J\}[BK]\{d_B = V, W' = d_K, d_B = d_K\}[CQ]\{d_C = W, d_Q = d_C\} + [OM]\{d_M = d_O, T' = d_M\}[FJ]\{d_F = T, d_J = d_F, V' = d_J\}[AQ]\{d_A = V, d_Q = d_A\})^*$.
The terms are indexed as:
$R_1 = ([OM]\{m^\ell = o^\ell, T^\ell = m^\ell\}[EN]\{e^\ell = T^\ell, U^\ell = n^\ell, e^\ell = n^\ell\}[HJ]\{h^\ell = U^\ell, V^\ell = j^\ell, j^\ell = h^\ell\}[BK]\{b^\ell = V^\ell, W^\ell =$

Table 1
Experimental results for the case studies 1–5.

Case studies	Reo components	Reo channels	Reo nodes	CA states	CA transitions	Computation time (s)
Alternator circuit	0	3	3	2	2	0.044
Nondeterministic choice	0	2	3	2	5	0.075
Fibonacci series	1	4	5	2	2	0.067
Banking transaction	3	14	12	5	9	0.053
Travel agency	8	18	15	5	9	0.023

Table 2
Experimental results for the travel agency case study with different numbers of components.

No. of hotels	No. of car rentals	No. of airlines	CA states	CA transitions	Memory usage (MB)	Computation time (s)
3	3	3	11	20	12.38	0.023
10	10	10	32	62	16.2	0.444
100	10	10	122	242	17	19
100	50	10	162	322	76.3	176
100	100	100	302	602	284.76	6780

$k^\ell, b^\ell = k^\ell\}[RD]\{d^\ell = W^\ell, r^\ell = d^\ell\} + [OM]\{m^\ell = o^\ell, T^\ell = m^\ell\}[EN]\{e^\ell = T^\ell, U^\ell = n^\ell, e^\ell = n^\ell\}[HJ]\{h^\ell = U^\ell, V^\ell = j^\ell, j^\ell = h^\ell\}[BK]\{b^\ell = V^\ell, W^\ell = k^\ell, b^\ell = k^\ell\}[CQ]\{c^\ell = W^\ell, q^\ell = c^\ell\} + [OM]\{m^\ell = o^\ell, T^\ell = m^\ell\}[EN]\{e^\ell = T^\ell, U^\ell = n^\ell, e^\ell = n^\ell\}[HJ]\{h^\ell = U^\ell, V^\ell = j^\ell, j^\ell = h^\ell\}[AQ]\{a^\ell = V^\ell, q^\ell = a^\ell\} + [OM]\{m^\ell = o^\ell, T^\ell = m^\ell\}[EN]\{e^\ell = T^\ell, U^\ell = n^\ell, e^\ell = n^\ell\}[GQ]\{g^\ell = U^\ell, q^\ell = g^\ell\} + [OM]\{m^\ell = o^\ell, T^\ell = m^\ell\}[FJ]\{f^\ell = T^\ell, j^\ell = f^\ell, V^\ell = j^\ell\}[BK]\{b^\ell = V^\ell, W^\ell = k^\ell, b^\ell = k^\ell\}[RD]\{d^\ell = W^\ell, r^\ell = d^\ell\} + [OM]\{m^\ell = o^\ell, T^\ell = m^\ell\}[FJ]\{f^\ell = T^\ell, j^\ell = f^\ell, V^\ell = j^\ell\}[BK]\{b^\ell = V^\ell, W^\ell = k^\ell, b^\ell = k^\ell\}[CQ]\{c^\ell = W^\ell, q^\ell = c^\ell\} + [OM]\{m^\ell = o^\ell, T^\ell = m^\ell\}[FJ]\{f^\ell = T^\ell, j^\ell = f^\ell, V^\ell = j^\ell\}[AQ]\{a^\ell = V^\ell, q^\ell = a^\ell\})^*$.
And finally we have:

$$r^\ell = o^\ell f^\ell b^\ell d^\ell \vee o^\ell h^\ell b^\ell d^\ell$$

$$q^\ell = o^\ell f^\ell a^\ell d^\ell \vee o^\ell h^\ell a^\ell d^\ell \vee o^\ell f^\ell b^\ell c^\ell \vee o^\ell h^\ell b^\ell c^\ell \vee o^\ell f^\ell a^\ell c^\ell \vee$$

$$o^\ell h^\ell a^\ell c^\ell \vee o^\ell g^\ell b^\ell d^\ell \vee o^\ell g^\ell b^\ell c^\ell \vee o^\ell g^\ell a^\ell d^\ell \vee o^\ell g^\ell a^\ell c^\ell.$$

This shows that for receiving a positive reply (r) for a request (o), we must have a car rental agency (b) airline (d), and at least one hotel (f or h) available.

Appendix C. Experimental results for the case studies

For each case study, in Table 1 we show the number of Reo components, channels, and nodes, plus the number of generated CA states and transitions, and the computation time for deriving the symbolic execution output. Table 2 shows the experimental results for the travel agency case study with different numbers of hotels, car rental agencies, and airlines. We show the numbers of CA states, transitions, memory usage and the computation time for each case. All measurements are performed on a notebook with an Intel core 2 Dou CPU T7250, 2GHz, and 2.00 GB memory, running Windows Vista.

References

[1] F. Arbab, Reo: a channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (3) (2004) 329–366.
[2] F. Arbab, C. Baier, J.J. Rutten, M. Sirjani, Modeling component connectors in Reo by constraint automata (extended abstract), in: Proceedings of the Second International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'03, in: Electronic Notes in Theoretical Computer Science, vol. 97, Elsevier, 2004, pp. 25–46.
[3] C. Baier, M. Sirjani, F. Arbab, J.J. Rutten, Modeling component connectors in Reo by constraint automata, Science of Computer Programming 61 (2006) 75–113.
[4] C. Koehler, A. Lazovik, F. Arbab, ReoService: coordination modeling tool, in: B.J. Krämer, K.-J. Lin, P. Narasimhan (Eds.), Proceedings of the Fifth International Conference on Service-Oriented Computing, ICSOC'07, Vienna, Austria, 2007, in: Lecture Notes in Computer Science, vol. 4749, Springer, 2007, pp. 625–626.
[5] A. Lazovik, F. Arbab, Using Reo for service coordination, in: B.J. Krämer, K.-J. Lin, P. Narasimhan (Eds.), Proceedings of the Fifth International Conference Service-Oriented Computing, ICSOC'07, Vienna, Austria, 2007, in: Lecture Notes in Computer Science, vol. 4749, Springer, 2007, pp. 398–403.
[6] S. Meng, F. Arbab, Web services choreography and orchestration in Reo and constraint automata, in: Proceedings of 22nd Annual ACM Symposium on Applied Computing, SAC'07, ACM, 2007, pp. 346–353.
[7] S. Tasharofi, M. Vakilian, R. Ziloochian, M. Sirjani, Modeling web services using coordination language Reo, in: Proceedings of the Fourth International Workshop on Web Services and Formal Methods, WS-FM'07, in: Lecture Notes in Computer Science, vol. 4937, Springer, 2007, pp. 108–123.

[8] S. Tasharofi, M. Sirjani, Formal modeling and conformance validation for WS-CDL using Reo and CASM, in: Proceedings of the Seventh International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'08, in: Electronic Notes in Theoretical Computer Science, vol. 159, Elsevier, 2008, pp. 99–115.
[9] F. Mahdikhani, M.R. Hashemi, M. Sirjani, QoS aspects in web services compositions, in: Proceedings of the Fourth IEEE International Symposium on Service-Oriented System Engineering, SOSE'08, IEEE, 2008, pp. 239–244.
[10] N. Kokash, F. Arbab, Applying Reo to service coordination in long-running business transactions, in: Proceedings of the 24st Annual ACM Symposium on Applied Computing, SAC'09, 2009, pp. 1381–1382.
[11] F. Arbab, N. Kokash, S. Meng, Towards using Reo for compliance-aware business process modeling, in: Margaria and Steffen [74], pp. 108–123.
[12] N. Razavi, M. Sirjani, Using Reo for formal specification and verification of system designs, in: Proceedings of the Fourth ACM-IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'06, IEEE, 2006, pp. 113–122.
[13] N. Razavi, M. Sirjani, Compositional semantics of system-level designs written in SystemC, in: Proceedings of the 2nd IPM International Symposium on Fundamentals of Software Engineering, FSEN'07, in: Lecture Notes in Computer Science, Springer, 2007, pp. 113–128.
[14] F. Arbab, L. Astefanoaei, F.S. de Boer, M. Dastani, J.-J.C. Meyer, N.A.M. Tinnemeier, Reo connectors as coordination artifacts in 2APL systems, in: Proceedings of the 11th Pacific Rim International Conference on Multi-Agents, PRIMA'08, in: Lecture Notes in Computer Science, Springer, 2008, pp. 42–53.
[15] D. Clarke, D. Costa, F. Arbab, Modelling coordination in biological systems, in: Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods, ISoLA'04, 2004, pp. 9–25.
[16] F. Arbab, N. Medvidovic, N. Mehta, M. Sirjani, Modeling behavior in compositions of software architectural primitives, in: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, ASE'04, 2004, pp. 371–374.
[17] Eclipse coordination tools home page, http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools.
[18] S. Klüppelholz, C. Baier, Symbolic model checking for channel-based component connectors, in: Proceedings of the Fifth International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'06, in: Electronic Notes in Theoretical Computer Science, Elsevier, 2006.
[19] C. Baier, T. Blechmann, J. Klein, S. Kluppelholz, A uniform framework for modeling and verifying components and connectors, in: Proceedings of 11th International Conference on Coordination Models and Languages, Coordination'09, in: Lecture Notes in Computer Science, vol. 5521, Springer, 2009, pp. 268–287.
[20] J.F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, M.V. Weerdenburg, The formal specification language mCRL2, in: Proceedings of the Dagstuhl Seminar, MIT Press, 2007.
[21] mCRL2, http://www.mcrl2.org.
[22] N. Kokash, C. Krause, E. de Vink, Data-aware design and verification of service composition with Reo and mCRL2, in: Proceedings of the 25th Annual ACM Symposium on Applied Computing, SAC'10, ACM Press, 2010, pp. 2406–2413.
[23] N. Kokash, C. Krause, E. de Vink, Time and data aware analysis of graphical service models in Reo, in: Proceedings of the IEEE International Conference on Software Engineering and Formal Methods, SEFM'10, IEEE Computer Society, 2010.
[24] J.C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.
[25] S. Khurshid, C.S. Pasareanu, W. Visser, Generalized symbolic execution for model checking and testing, in: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'03, in: Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 553–568.
[26] J.A. Brzozowski, Derivatives of regular expressions, Journal of ACM 11 (4) (1964) 481–494.
[27] C. Gaston, P.L. Gall, N. Rapin, A. Touil, Symbolic execution techniques for test purpose definition, in: Proceedings of the 18th International Conference on Testing Communicating Systems, TestCom'06, 2006, pp. 1–18.
[28] C.S. Pasareanu, W. Visser, Verification of java programs using symbolic execution and invariant generation, in: Proceedings of the 11th International SPIN Workshop on Model Checking of Software, SPIN'04, 2004, pp. 164–181.
[29] S. Anand, C.S. Pasareanu, W. Visser, Symbolic execution with abstraction, International Journal on Software Tools for Technology Transfer (STTT) 11 (1) (2009) 53–67.
[30] N. Kokash, C. Krause, E. de Vink, Verification of context-dependent channel-based service models, in: Proceedings of the International Symposium on Formal Methods for Components and Objects, FMCO'09, in: Lecture Notes in Computer Science, Springer, 2010.
[31] M.R. Mousavi, M. Sirjani, F. Arbab, Formal semantics and analysis of component connectors in Reo, in: Proceedings of the Fourth International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'05, in: Electronic Notes in Theoretical Computer Science, vol. 154, Elsevier, 2006, pp. 83–99.
[32] B. Pourvatan, M. Sirjani, H. Hojjat, F. Arbab, Automated analysis of Reo circuits using symbolic execution, in: Proceedings of the Eighth International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'09, in: Electronic Notes in Theoretical Computer Science, vol. 255, Elsevier, 2009, pp. 137–158.
[33] F. Arbab, Composition of interacting computations, in: D. Goldin, S.A. Smolka, P. Wegner (Eds.), Interactive Computation, Springer, 2006, pp. 277–321.
[34] G.A. Papadopoulos, F. Arbab, Coordination models and languages, Advances in Computers 46 (1998) 330–401.
[35] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems TOPLAS (1) (1985) 80–112.
[36] A.W. Colman, J. Han, Coordination systems in role-based adaptive software, in: Proceedings of the 7th International Conference on Coordination Models and Languages, Coordination'05, in: Lecture Notes in Computer Science, vol. 3454, Springer, 2005, pp. 63–78.
[37] F. Arbab, The IWIM model for coordination of concurrent activities, in: Proceedings of the First International Conference on Coordination Models and Languages, Coordination'96, in: Lecture Notes in Computer Science, vol. 1061, Springer, 1996, pp. 34–56.
[38] J.C. Cruz, S. Ducasse, A group based approach for coordinating active objects, in: Proceedings of the Third International Conference on Coordination Models and Languages, Coordination'99, in: Lecture Notes in Computer Science, vol. 1594, Springer, 1999, pp. 355–370.
[39] A. Omicini, F. Zambonelli, Tuple centres for the coordination of Internet agents, in: Proceedings of the ACM Symposium on Applied Computing, 1999, pp. 183–190.
[40] A. Omicini, E. Denti, Formal ReSpecT, in: Declarative Programming—Selected Papers from AGP'00, in: Electronic Notes in Theoretical Computer Science, vol. 48, Elsevier, 2001, pp. 179–196.
[41] A. Omicini, Formal ReSpecT in the A&A perspective, in: Proceedings of the Fifth International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'06, in: Electronic Notes in Theoretical Computer Science, vol. 175, Elsevier, 2007, pp. 97–117.
[42] C. Talcott, M. Sirjani, S. Ren, Comparing three coordination models: Reo, ARC, and RRD, in: Proceedings of the Sixth International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA'07, in: Electronic Notes in Theoretical Computer Science, vol. 194, Elsevier, 2008, pp. 39–55.
[43] C. Talcott, M. Sirjani, S. Ren, Comparing three coordination models: Reo, ARC, and RRD, Science of Computer Programming, Special issue of FOCLASA'07.
[44] S. Ren, Y. Yu, N. Chen, K. Marth, P.-E. Poirot, L. Shen, Actors, roles and coordinators: a coordination model for open distributed and embedded systems, in: Proceedings of 8th International Conference on Coordination Models and Languages, Coordination'06, in: Lecture Notes in Computer Science, vol. 4038, Springer, 2006, pp. 247–265.
[45] J. Meseguer, C.L. Talcott, Semantic models for distributed object reflection, in: Proceedings of the 16th European Conference on Object-Oriented Programming, ECOOP'2002, in: Lecture Notes in Computer Science, vol. 2374, Springer, 2002, pp. 1–36. invited paper.
[46] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, USA, 1990.
[47] M. Sirjani, M.-M. Jaghoori, C. Baier, F. Arbab, Compositional semantics of an actor-based language using constraint automata, in: Proceedings of the Eighth International Conference on Coordination Models and Languages, Coordination'06, in: Lecture Notes in Computer Science, vol. 4038, Springer, 2006, pp. 281–297.

[48] D. Jordan, J. Evdemon, Web Services Business Process Execution Language Version 2.0, August 2006.
[49] Web services choreography description language version 1.0, W3C Candidate Recommendation, http://www.w3.org/TR/ws-cdl-10/, November 2005.
[50] F. Arbab, J. Rutten, A coinductive calculus of component connectors, in: Proceedings of the 16th International Workshop on Recent Trends in Algebraic Development Techniques, WADT'02, in: Lecture Notes in Computer Science, vol. 2755, 2002, pp. 34–55.
[51] F. Arbab, CASM: constraint automata with state memory, unpublished notes.
[52] L.A. Clarke, A system to generate test data and symbolically execute programs, IEEE Transactions on Software Engineering 2 (3) (1976) 215–222.
[53] R.S. Boyer, B. Elspas, K.N. Levitt, Select—a formal system for testing and debugging programs by symbolic execution, SIGPLAN Notices 10 (6) (1975) 234–245.
[54] J. Zhang, C. Xu, X. Wang, Path-oriented test data generation using symbolic execution and constraint solving techniques, in: Proceedings of the Second International Conference on Software Engineering and Formal Methods, SEFM'04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 242–250.
[55] J. Zhang, Constraint solving and symbolic execution, in: Verified Software: Theories, Tools, Experiments, in: Lecture Notes in Computer Science, vol. 4171, Springer, 2008, pp. 539–544.
[56] A.R. Bradley, Z. Manna, The Calculus of Computation: Decision Procedures with Applications to Verification, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
[57] R.Y. Kain, Automata Theory: Machines and Languages, Robert E. Krieger Publishing Company, 1981.
[58] C. Neumann, Converting deterministic finite automata to regular expressions, 2005.
[59] D.N. Arden, Delayed-logic and finite-state machines, Theory of Computing Machine Design (1960) 1–35.
[60] Z. Manna, A. Pnueli, Temporal Verification of Reactive Systems (Safety), Springer-Verlag, Berlin, Germany, 1995.
[61] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, The MIT Press, Cambridge, Massachusetts, 1999.
[62] E.A. Emerson, Temporal and modal logic, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier Science Publishers, Amsterdam, 1990, pp. 996–1072.
[63] K. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.
[64] C. Baier, J.-P. Katoen, Principles of Model Checking, The MIT Press, 2008.
[65] C.S. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, M. Pape, Combining unit-level symbolic execution and system-level concrete execution for testing NASA software, in: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, ACM, 2008, pp. 15–26.
[66] M. Pandey, R. Raimi, R.E. Bryant, M.S. Abadir, Formal verification of content addressable memories using symbolic trajectory evaluation, in: Proceedings of the 34th annual conference on Design automation, DAC'97, ACM, 1997, pp. 167–172.
[67] R.E. Bryant, D.L. Beatty, C.-J.H. Seger, Formal hardware verification by symbolic ternary trajectory evaluation, in: Proceedings of the 28th conference on ACM/IEEE design automation, DAC'91, ACM, 1991, pp. 397–402.
[68] C.-J.H. Seger, R.B. Jones, J.W. O'Leary, T.F. Melham, M. Aagaard, C. Barrett, D. Syme, An industrially effective environment for formal hardware verification, IEEE Transactions on CAD of Integrated Circuits and Systems 24 (9) (2005) 1381–1405.
[69] S. Morasca, M. Pezzè, Validation of concurrent ADA programs using symbolic execution, in: Proceedings of the 2nd European Software Engineering Conference, ESEC'89, Springer-Verlag, 1989, pp. 469–486.
[70] S.F. Siegel, A. Mironova, G.S. Avrunin, L.A. Clarke, Combining symbolic execution with model checking to verify parallel numerical programs, ACM Transactions on Software Engineering and Methodology 17 (2) (2008) 1–34. Article 10.
[71] N. Lynch, M. Tuttle, An introduction to input/output automata, CWI Quarterly 2 (3) (1989) 219–246.
[72] F. Arbab, S. Meng, Synthesis of connectors from scenario-based interaction specifications, in: M.R.V. Chaudron, C.A. Szyperski, R. Reussner (Eds.), CBSE, in: Lecture Notes in Computer Science, vol. 5282, Springer, 2008, pp. 114–129.
[73] F. Arbab, N. Kokash, S. Meng, Towards using reo for compliance-aware business process modeling, in: Margaria and Steffen [74], pp. 108–123.
[74] T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13–15, 2008, Proceedings, in: Communications in Computer and Information Science, vol. 17, Springer, 2008.