Modeling and Analysis of Reo Connectors Using Alloy

Ramtin Khosravi[1], Marjan Sirjani[1,2], Nesa Asoudeh[1], Shaghayegh Sahebi[1], and Hamed Iravanchi[1]

[1] School of Electrical and Computer Engineering, University of Tehran, Kargar Ave., Tehran, Iran
[2] School of Computer Science, Institute for Studies in Theoretical Physics and Mathematics, P.O. Box: 19395-5746, Tehran, Iran
rkhosravi@ece.ut.ac.ir, msirjani@ut.ac.ir, {n.asoudeh,s.sahebi,h.iravanchi}@ece.ut.ac.ir

Abstract. Reo is an exogenous coordination language based on a calculus of channel composition. Different formal models have been developed for this language. In this paper, we present a new approach to modeling and analysis of Reo connectors using Alloy which is a lightweight modeling language based on first-order relational logic. We provide a reusable library of Reo channels in Alloy that can be used to create a model of a Reo connector in Alloy. The model is simple and reflects the original structure of the connector. Furthermore, the model of a connector can be reused as a component for constructing more complex connectors. Using the Alloy Analyzer tool, properties expressed as predicates can be verified by automatically analyzing the execution traces of the Reo connector. We handle the context-sensitive behavior of channels as well as optional constraints on the interactions with environment. Our compositional model can be used as an alternative to other existing approaches, and is supported by a well known tool with a rich set of features such as counterexample generation.

1 Introduction

The concept of "coordination" has resulted to a new class of models, formalisms and mechanisms for describing concurrent and distributed computations. A coordination language used to develop a coordination model is able to integrate a set of possibly heterogeneous components together [15]. Reo is a coordination language based on components and connectors [2]. It offers a way for compositional construction of systems out of black-box components. To achieve this, Reo provides a compositional mechanism to construct various connectors (commonly known as Reo *circuits*). This is based on the notion of *channel* as primitive connectors from which more complex ones are constructed. We will review basic concepts of Reo in Sect. 3.1.

As the main use of coordination languages is in modeling reactive and concurrent systems, the problem of verification of models becomes an important issue, because in these systems correctness is much harder to verify manually, or with testing. Our goal is to provide a method to model and analyze Reo connectors using Alloy, which is a lightweight modeling language, based on first-order relational logic [8]. It is supported by an efficient tool called Alloy Analyzer [1] that will serve as the analysis tool in our method. We briefly review the Alloy modeling language in Sect. 3.2.

The basic idea of our method is to model the behavior of a Reo connector by the set of all execution traces of the connector. Each trace is a bounded sequence of states. In our method, the structure of a Reo connector is constructed compositionally, from smaller connectors (which are ultimately basic channels). Each channel imposes a constraint on the states of the trace. Other constraints are the facts that specify the behavior of the environment of the circuit. We will explain our modeling method in more detail in Sect. 4. For the sake of simplicity, our method ignores the values of data passed along Reo channels, and just considers data flow, though it can be extended to handle data values too.

The modular structure of Alloy allows us to separate the definitions of primitive Reo constructs from the description of a circuit. We have provided the definitions of a set of Reo connectors in a module named Relloy [16] that can be used as a reusable library when modeling various circuits. In Sect. 5.1, we will see how to describe a Reo circuit using our method. To analyze a Reo circuit, the modeler provides the properties to be checked in terms of first-order predicate logic formulas, and the Alloy Analyzer automatically checks the properties using a SAT solver. This way of analysis is not "model checking" and is based on checking all execution traces of at most a specified number of states. The traces are generated automatically, satisfying the constraints imposed by the channels and other facts in the model. Analyzing Reo circuits in our model will be more elaborated in Sect. 5.2, with a case study on a round-robin dispatcher [17]. We also address the context-sensitive behavior of connectors, by imposing maximal progress property on the traces (Sect. 6).

The benefits of our method can be summarized in the following items:

Coverage of Semantics: As mentioned above, we handle all basic concepts of Reo, as well as the aspects which are normally harder to address, such as context-sensitive behavior and modeling environment. All of these are based on the sound basis of relational logic.

Compositionality: The model of Reo connectors are constructed compositionally from smaller ones easily. Issues such as renaming are handled automatically by the block structures of Alloy at language level.

Clarity: The description of a Reo connector in our model clearly represents the original structure of the circuit. Since it does not involve complex mathematical notations, it is more familiar to software engineers. As the syntax of the description is textual, it is much easier to work with when modeling large models.

Tool Support: Alloy is a well-known language, with an efficient tool support. It offers useful features such as counterexample generation and visualization which can help the modeler to model and debug the connector in small iterations.

As some of the benefits above are inherited from Alloy, our method has got some of its limitations too. The main limitation is that our traces are bounded, so if Alloy cannot find a counterexample, it does not mean there is none. The assumption here is that most flaws can be discovered when considering all possible traces within small bounds exhaustively. Another problem is scalability, as the analysis takes some time when the model becomes large. However, in an ongoing work, we are trying to improve the efficiency by techniques that will be mentioned later. In Sect. 2, we briefly review existing approaches to model Reo semantics.

2 Related Work

Different formal semantics have been developed for Reo, including Timed Data Streams (TDS) [3], Constraint Automata (CA) [4], Graph Coloring [5], and Structural Operational Semantics (SOS) [14]. Timed Data Streams model the possible flows of data on connector ports, assigning a time to each interaction (input or output of a data element). The declarative and relational nature of this semantics is one of its strengths; but there is no support for simulation or model checking. Constraint automata is a compositional and operational semantics for Reo. Constraint automata shall be extended with priorities to capture the context sensitive behavior of connectors. Also, the interaction with environment is not modeled and I/O requests are considered always available. A symbolic model checker based on CA semantics is developed [11] and CTL-like properties can be checked.

The idea of graph coloring semantics is marking data flow or its absence by colors. A coloring table for a Reo connector actually describes the possible behavior in a particular configuration of the connector, which includes the states of channels, plus the presence or absence of I/O requests. A coloring corresponds to a possible next step based on that configuration. Here, input and output operations are considered as primitives as well as channels and nodes. A join operation is then defined on coloring tables. To capture the context sensitive behavior of connectors a third color shall be added to the semantics. The goal of graph coloring is more to build a basis for distributed implementation of Reo circuits despite of generating a semantics basis for analysis.

Based on the Structural Operational Semantics (SOS) of Reo a model checker using Maude system [13] is developed.

Our approach is similar to CA in its simple way of modeling the Reo connectors and similar to graph coloring in capturing the behavior by recognizing the blocking nodes and synchronous clusters, to obtain the transitions. Furthermore, our method simply models the behavior of Reo connectors including context sensitive behavior and I/O requests of environment without additional complexity. Also, it provides the support of the already-exist automated analyzer of Alloy.

3 Preliminaries

In this section, we briefly review basic concepts of Reo coordination language and Alloy modeling language. Parts of our discussion on Reo are taken from [4] and [2] and the overview of Alloy is mainly based on [9].

3.1 Reo

Reo is a coordination language based on components and connectors. As coordination is the main point of concern, the emphasis of Reo is on connectors. A connector is presented as a graph of nodes and edges where edges represent channels and nodes are channel ends. Primitive connectors in Reo are called channels which provide the basic communication mechanisms. A channel always has two ends. There are two types of channel ends: source and sink. The source end is the point where data enters into the channel. The sink end is the point where data leaves the channel. Note that both channel ends may be of the same type (both source or both sink). Channel ends are connected to each other via nodes. If all channel ends adjacent to a node are source ends (resp. sink ends), we call the node a source node (resp. sink node). If there are channel ends of both types, then the node is called a mixed node.

Reo provides operations that enable components to perform I/O on nodes. 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 component can obtain data items from a sink node that it is connected to through destructive take input operations. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if there are more, one is selected nondeterministically. A mixed node combines the behavior of a sink node and a source node.

There are several mixed node types in Reo as indicated in Fig. 1. The data items simply flow through a *flow through* node. A *merge node* delivers a value out of one of the incoming channels nondeterministically. A write on a *replication node* succeeds only if all outgoing channels are capable of consuming the written data. We say a node *can be fired*, if it can successfully pass the data according to the mentioned rules.

There are different channel types in Reo. Each channel passes data in a predefined direction. Structurally, each unidirectional channel has a source end that receives data and a sink end that dispenses it. Bidirectional channels have either two source ends, or two sink ends. There are no 'fixed' set of channel types in Reo, and new ones can be defined freely with their own policy for synchronization, buffering, computations, etc.

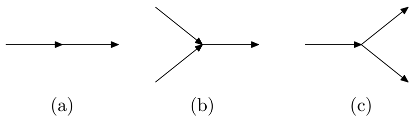

Fig. 1. Three types of mixed nodes: (a) flow through, (b) nondeterministic merge, and (c) replication

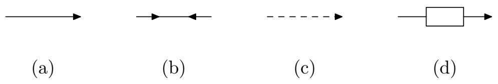

Fig. 2. Channel notations: (a) Sync (b) SyncDrain (c) LossySync (d) FIFO1

Here, we define the channel types that have been used throughout the examples in this paper (Fig. 2). The definition for more channel types can be found in [2]. Note that our reusable library of channels contain more channel types that are presented in this paper and includes most widely used channels types.

The simplest channel is synchronous (Sync) channel that has a source and a sink end, and no buffer. It accepts a data item through its source end if and only if it can simultaneously dispense it through its sink. A synchronous drain (SyncDrain) accepts data items from its both ends simultaneously and the data values will be lost. A lossy synchronous (LossySync) channel is similar to a Sync channel, except that it always accepts all data items through its source end. If it is possible for it to simultaneously dispense the data item through its sink the channel transfers the data item; otherwise the data item is lost. A FIFO1 channel represents an asynchronous channel with a buffer of capacity one. A write operation on the source end succeeds only if the buffer is empty, and a take operation on the sink end succeeds only if the buffer contains data. This buffer may be initially empty or contain some data item.

3.2 Alloy

Alloy is a lightweight modeling language based on the first order relational logic [10]. In an Alloy model, there are a number of *signatures* each defining a set of atoms. The definition of a signature may contain a number of *fields* which define *relations* between atoms of signatures. Signatures also serve as types, and subtyping is possible through signature extension. There are also ways to define constraints in the model, using constraint paragraphs. There are four kinds of constraint paragraphs:

Fact: A constraint that always holds

Predicate: Named and parameterized formulas that can be used elsewhere

Function: Named and parameterized expressions that can be used elsewhere

Assertion: A constraint that is intended to follow from the facts of a model

Note that facts can be defined in two ways: either following a signature declaration, or elsewhere in the model. In the first case, they are implicitly quantified over all atoms of the signature.

A model in Alloy means a collection of *instances*. Instances are binding of values to variables. Alloy Analyzer finds instances of a model automatically by assigning values to variables satisfying the constraints defined. Model analysis involves constraint solving, and the analyzer embodies a SAT solver. It provides visualization for making sense of solutions and counterexamples it finds [9].

Instructions to Alloy Analyzer to perform its analysis are called *commands*. A *run* command causes the analyzer to search for an instance that witnesses the consistency of a function or a predicate. A *check* command causes it to search for a counterexample to show that an assertion does not hold. Searching for instances is done within finite bounds, specified by the user as *scope*. So, when the search fails, it does not mean that there is no instance satisfying the model (i.e., the model is inconsistent). Alloy analysis is based on *small scope hypothesis*, which says that if we consider all small instances, most flaws will be revealed [10].

In this paper, when writing Alloy definitions and formulas, we sometimes use common mathematical symbols instead of Alloy keywords (e.g. ∀ or ∈ symbols instead of 'all' or 'in' keywords).

4 Modeling Basic Reo Constructs in Alloy

In this section, we show how primitive constructs in Reo, nodes, channels and more complex connectors, are modeled in Alloy. We first see how Reo connectors can be constructed from simpler ones structurally, and then study the behavioral modeling of connectors. Note that we defer handling context-sensitive behavior to Sect. 6.

4.1 Modeling Connector Structure

The basic goal of Reo is to formalize the concept of connectors which serve as the pathways through which components communicate. A connector in turn, is composed of simpler ones, which are ultimately composed of channels. A component is connected to a connector through a number of externally visible nodes which we call *ports*. The two basic concepts in the model are *nodes* and *connectors* that are represented as two Alloy signatures as follows:

```
sig Node {}
abstract sig Connector {
    conns : set Connector,
    ports : set Node
}
```

According to the above definitions, a node is an atomic concept, but connectors may be constructed recursively from other connectors. The signature *Connector* is marked abstract as concrete connectors will be defined as its subsignatures. For a connector *c*, *c.conns* represents the set of connectors that are parts of *c*. This model of constructing connectors resembles the *Composite* design pattern [7]. The set *c.ports* is the set of nodes of *c* that are accessible from outside (i.e., the nodes of c that are not hidden). These nodes can be attached to the ports of other connectors.

Note that in Alloy, a field of a signature is considered as a relation from that signature to the type of the field. For example, *conns* is a relation that relates each instance of *Connector* to a set of *Connectors* (its parts). So, the notations *c.conns* and *conns*[*c*] are equivalent.

The simplest form of a connector is a channel, defined by the following signature:

```
abstract sig Channel extends Connector {
    $e_1, e_2$ : one Node
}
{ $(ports = e_1 + e_2) \wedge (conns = \emptyset)$ }
```

Each *Channel* has two fields e_1 and e_2, each of them is a reference to a single *Node*. The signature is defined abstract, as it will be refined later into specific channels, but common to all channel are two facts: they have two ports (as the channel ends), and they are atomic connectors, so the *conns* field is the empty set. The expression $e_1 + e_2$ denotes the set $\{e_1, e_2\}$. Each specific channel type has a separate signature extending *Channel*. Normally, the channel types do not add any 'structural' feature to the definition of abstract *Channel*. We will consider the behavior of channels later.

```
sig Sync extends Channel {}
sig Drain extends Channel {}
sig Lossy extends Channel {}
sig Fifo extends Channel {}
```

Another simple connector defined is a *Merger* which is used to model merge nodes with two inputs in Reo. A merger has references to two nodes i_1 and i_2 as its inputs and a third node o as its output node:

```
abstract sig Merger extends Connector {
    disj $i_1, i_2, o$ : one Node
}
{ $(ports = i_1 + i_2 + o) \wedge (conns = \emptyset)$ }
```

To illustrate how a composite connector is constructed, we define a simple connector composed of a FIFO1 channel attached to the end of a synchronous channel (Fig. 3).

```
sig SyncFifo extends Connector {
    $a, b, d$ : one Node,
    $s$ : one Sync, $f$ : one Fifo
} {
    $a = s.e_1$
    $d = f.e_2$
    $b = s.e_2 \wedge b = f.e_1$
    $(conns = s + f) \wedge (ports = a + d)$
}
```

Note that there are three nodes in an instance of *SyncFifo*, but only two of them (a and d) comprise the ports of the connector, and the other one (b) is 'hidden'. The *SyncFifo* connector defined above can be used in constructing more complex connectors in turn. We will see an example in 5.1.

The above definitions do not constrain the instances to form valid Reo circuits. A few more facts are required to ensure this. The first one states that a node that is hidden in a connector cannot be referenced in the enclosing connectors:

fact $\{\ \forall\ c : \mathit{Connector} \mid \nexists\ n : \mathit{hiddens}[c] \mid n \in \mathit{nodes}[^\wedge \mathit{conns}.c]\ \}$

The expression $\mathit{hiddens}[c]$ denotes the set of all hidden nodes of c and $\mathit{nodes}[^\wedge \mathit{conns}.c]$ is the set of all nodes in all connectors that are directly or indirectly contain c ($^\wedge \mathit{conns}$ is the transitive closure of *conns* relation). We have omitted the definition of the two functions *hiddens* and *nodes* for brevity.

The other facts constrain the composition of the connectors to form a rooted tree. First, we define a singleton signature *Circuit* having a reference to one root connector:

one sig *Circuit* { *root* : **one** *Connector* }

The first fact below states that the *conns* relation is acyclic, and the two others define *Circuit.root* as the root of the tree:

```
fact {
    $\nexists\ c : \mathit{Connector} \mid c \in c.^\wedge \mathit{conns}$
    $\forall\ c : \mathit{Connector} - \mathit{Circuit.root} \mid c \in \mathit{Circuit.root}.^\wedge \mathit{conns}$
    $\nexists\ c : \mathit{Connector} \mid \mathit{Circuit.root} \in c.\mathit{conns}$
}
```

4.2 Modeling Connector Behavior

To model the behavior of a Reo connector, we use traces of computation which is a common technique in Alloy. For a Reo circuit, we define a *trace* of computation as a sequence of *states*. In each state, we record the state of FIFO1 buffers as well as the set of nodes that are to be fired to go to the next state:

```
sig State {
    fire : set Node,
    full : set Fifo
}
```

The set *fire* is the set of nodes that are to be fired at that state and *full* denotes the set of FIFO1 channels with full buffer.

The notion of state in our method is close to the notion of state in constraint automata. For example, in Fig. 3(a) and 3(b), a simple circuit and its corresponding constraint automaton are shown. Figure 3(c) shows the first five states of an execution trace as used in our method. Assuming the environment is always ready to perform write and take operations on nodes a and d respectively, in states $T_1, T_3, \ldots$, we have $\mathit{fire} = \{a, b\}, \mathit{full} = \{\}$ and in states $T_2, T_4, \ldots$, we have $\mathit{fire} = \{d\}, \mathit{full} = \{f\}$.

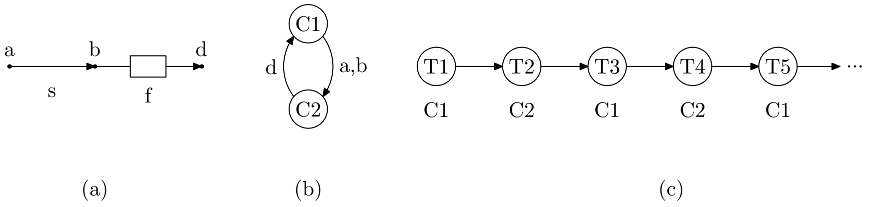

Fig. 3. (a) A simple Reo circuit (b) The corresponding constraint automaton with two states C_1 (empty buffer) and C_2 (full buffer) (c) An execution trace of the circuit – the states T_{2k+1} are corresponding to C_1 and the states T_{2k} are corresponding to C_2

To model a trace in Alloy, we have reused a standard module *util/ordering* which creates a single linear ordering over the atoms of the signature provided as its input (in our case, *State*). We use the *first* and *last* symbols defined in *util/ordering* to refer to the first and last states of a trace respectively. Note that since Alloy searches for model instances within a bounded scope, the traces are always bounded.

To model the behavior of each channel, we provide a fact that puts a constraint on the behavior of the whole circuit. For example, the behavior of the three simple (stateless) channels is modeled by the following three facts:

```
fact {
    $\forall\ s : State, c : Sync \mid c.e_1 \in s.fire \Leftrightarrow c.e_2 \in s.fire$
    $\forall\ s : State, c : Drain \mid c.e_1 \in s.fire \Leftrightarrow c.e_2 \in s.fire$
    $\forall\ s : State, c : Lossy \mid c.e_2 \in s.fire \Rightarrow c.e_1 \in s.fire$
}
```

Note that since we have ignored the actual data values, the facts for Sync and SyncDrain is the same.

The behavior of a merger connector is defined by the following fact:

```
fact {
    $\forall\ s : State, m : Merger \mid \{$
        $\neg\ (m.i_1 \in s.fire \wedge m.i_2 \in s.fire)$
        $m.o \in s.fire \Leftrightarrow (m.i_1 \in s.fire \vee m.i_2 \in s.fire)$
    }
}
```

The behavior of a FIFO1 channel relates two subsequent states of a trace together. In the following fact, *next*[*s*] denotes the next state in the trace, and *last* denotes the last state of the trace:

```
fact {
    $\forall\ s : State - last, c : Fifo \mid \{$
        $c.e_2 \in s.fired \Rightarrow (c \in s.full \wedge c \notin next[s].full)$
        $c.e_1 \in s.fired \Rightarrow (c \notin s.full \wedge c \in next[s].full)$
        $(c.e_1 \notin s.fired \wedge c.e_2 \notin s.fired) \Rightarrow (c \in s.full \Leftrightarrow c \in next[s].full)$
    }
}
```

A primary task of Alloy Analyzer is to assign a value to the *fire* set for each state such that the facts corresponding to the channels are satisfied. This automatically handles nondeterminism in selecting merge inputs.

4.3 Modeling Environment

To analyze the behavior of the circuit, we must be able to specify the behavior of the environment (i.e, the components attached to the ports of the root connector). To do this, we add a field *env_ready* to *State* to specify if the environment is ready to perform read/take operations on the ports of the root connector.

```
sig State {
    ...
    env_ready : set Node
} {
    env_ready ⊆ Circuit.root.ports
    fire ∩ ports ⊆ env_ready
}
```

The first fact constrains the set *env_ready* to contain only ports of the top-most connector, and the second one states that only ready ports can be fired.

The modeler can now provide facts to specify how the environment behaves. Alloy Analyzer automatically assigns values to the set *env_ready* satisfying the given facts when generating all possible traces.

5 Modeling and Analyzing Reo Circuits

In this section, we show the modeler's view of our method, that is how a circuit in Reo can be described and analyzed using the primitives explained in Sect. 4. The modular structure of Alloy allows us to define the basic Reo primitives in a separate module (we call it Relloy), and let the modules containing circuit descriptions import those definitions. This way, the circuit module only contains the description of the circuit along with the properties to be analyzed.

5.1 Describing a Reo Circuit

A Reo circuit is modeled as a signature extending *Connector*, the same way as defined *SyncFifo* in Sect. 4.1 In this section, we study how a complex circuit can be composed of simpler connectors. Consider the circuit in Fig. 4(a) which dispatches data from node a to nodes i, j, and k in a round-robin fashion. Instead of directly modeling the circuit, we use three instances of a simpler part connected together (Fig. 4(b)).

The complete description of the above connectors in our method is shown in Fig. 5. The signature *RRPart* models the connector in Fig. 4(b) and *RR* models the whole dispatcher.

The fact paragraph at the end contains two facts. The first fact states that the circuit root to be analyzed is a connector of type *RR*. The second fact determines the set of full FIFO1 channels in the initial state.

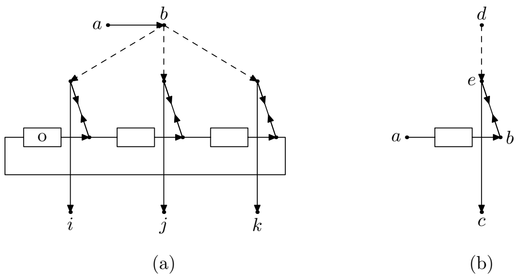

Fig. 4. (a) A round-robin 3-dispatcher circuit (b) A connector used to construct round-robin dispatcher

```
module RoundRobin
open Relloy

sig RRPart extends Connector {
  disj a, b, c, d, e : one Node,
  f : one Fifo,
  r : one Drain,
  l : one Lossy,
  s : one Sync
} {
  conns = f + r + s
  ports = a + b + c + d
  fifo[a, f, b]
  drain[b, r, e]
  lossy[d, l, e]
  sync[e, s, c]
}
sig RR extends Connector {
  disj a, b, i, j, k : one Node,
  disj c1, c2, c3 : one RRPart,
  s : one Sync
} {
  c1.a = c3.b
  c2.a = c1.b
  c3.a = c2.b
  c1.c = i
  c2.c = j
  c3.c = k
  c1.d = b
  c2.d = b
  c3.d = b
  sync[a, s, b]
  conns = c1 + c2 + c3 + s
  ports = a + i + j + k
}

fact {
  Circuit.root in RR
  first.full = Circuit.root.c1.f
}
```

Fig. 5. The description of a round-robin three-dispatcher connector in Alloy

5.2 Analyzing Circuits

To check the correctness of a circuit, we can express the desired properties in terms of assertions to be checked by the Alloy Analyzer. To provide an assertion, we can write formulas on the states of nodes and buffers in different states of the execution trace. For example, to express that in every state, only one of the buffers is full, we can write the following assertion (the # operator returns the size of its operand):

```
assert one_full { all s : State | #s.full = 1 }
```

The following command makes Alloy Analyzer search for a counterexample in all possible traces with maximum 10 states:

```
check one_full for 10 State
```

In this case, the response of Alloy Analyzer would be '`No counterexample found. Assertion may be valid.`' indicating the property holds for all traces of length at most 10. But if we mistakenly had `c3.a = c1.b` instead of `c3.a = c2.b` in the description of *RR*, we would get the response '`Counterexample found. Assertion is invalid.`'. By inspecting the given counterexample, we can find out the source of the flaw easily.

As another example, assume that we want to check that the data items written to a are never lost. This means that if a is to be fired in a state, then one of i, j, or k must be fired in the same state. This is expressed by the following assertion:

$$\forall s : \mathit{State} \mid a \in s.\mathit{fire} \Rightarrow \{i, j, k\} \cap s.\mathit{fire} \neq \emptyset$$

Again, we get a counterexample exposing a case in which the components attached to the ports i, j, and k are slower than the component writing at a. Hence, there is a state in which node a is ready while none of the three other ports are ready. In that case, the data written to a will be lost by all three LossySync channels. To fix this, we must replace the three LossySync channels with a three-way exclusive router[4], connecting b to node e of *RRPart* connectors. We have made this change, by defining a separate connector for exclusive router (and removing the LossySync channel from *RRPart*) and has successfully checked the above property. Due to lack of space, we do not present the Alloy descriptions here.

6 Handling Context-Sensitive Behavior

In this section, we show how to extend our model to handle the behavior exposed by channels like LossySync which is called context-sensitive behavior. An example is shown in figure 6.

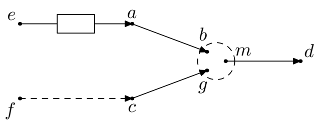

Fig. 6. An example of a Reo circuit demonstrating context-sensitive behavior

Consider the situation in which the environment is ready to write to both nodes e and f and is ready to take from b. Node e accepts the data item and writes it into the buffer. Considering the LossySync $f \dashrightarrow c$ in isolation, it may pass the data item to c or lose it. But, the *maximal progress* property of Reo requires it to always pass the data, because nodes a and b cannot be fired, and the merger input g can be fired. Note that the maximal progress is a global property, and cannot be enforced by adding some constraints to the behavioral description of the channel types locally.

To handle these cases, we find which nodes *cannot* be fired in a state, and force every other node to be fired in that state. In each state, some nodes may be locally detected as *blocked*, i.e., they cannot be fired. This may happen in four cases: the first endpoint of a full FIFO1, the second endpoint of an empty FIFO1, the input of a merge that is not selected, and an external port that is not ready.

```
pred $\mathit{blocked}\ [s : \mathit{State},\ n : \mathit{Node}]$ {
    $(\exists\ c : \mathit{Fifo} \mid (c \in s.\mathit{full} \wedge n = c.e_1) \vee (c \notin s.\mathit{full} \wedge n = c.e_2)) \vee$
    $(\exists\ m : \mathit{Merger} \mid (n = m.i_1 \vee\ n = m.i_2) \wedge n \notin s.\mathit{fire})$
    $n \in \mathit{Circuit.root.ports} - s.\mathit{env\_ready}$
}
```

As an example, in Fig. 6, and under the assumption that the environment is ready to write into e and f and read from b, a is the only blocked node. When a node is blocked, it may block other nodes from being fired, for example, through channels with synchronous behavior. For each *State* we define a relation $\mathit{can\_block} : \mathit{Node} \rightarrow \mathit{Node}$, such that $(m, n) \in \mathit{can\_block}$ means that in that state, if m is blocked, then n is blocked too. This may be caused by the existence of a channel with synchronous nature between m and n, like Sync, SyncDrain, or LossySync. Another case happens between the output node of merger and its selected input. It is important to not include this relation for the non-selected input, as it may incorrectly block some other nodes in the circuit. This fact is the reason for defining *can_block* as a field of *State*, so that it is computed for each state separately.

```
sig $\mathit{State}$ {
    $\mathit{fire}$ : set $\mathit{Node}$,
    $\mathit{full}$ : set $\mathit{Fifo}$,
    $\mathit{can\_block} : \mathit{Node} \rightarrow \mathit{Node}$
} {
    $\forall\ m, n : \mathit{Node} \mid m \rightarrow n \in \mathit{can\_block} \Leftrightarrow$ {
        $(\exists\ c : \mathit{Sync} \mid (c.e_1 = m \wedge c.e_2 = n) \vee (c.e_1 = n \wedge c.e_2 = m)) \vee$
        $(\exists\ c : \mathit{Drain} \mid (c.e_1 = m \wedge c.e_2 = n) \vee (c.e_1 = n \wedge c.e_2 = m)) \vee$
        $(\exists\ c : \mathit{Lossy} \mid (c.e_1 = m \wedge c.e_2 = n)) \vee$
        $(\exists\ c : \mathit{Merger} \mid (m \in \mathit{fire} \wedge \mathit{input}[c, m] \wedge \mathit{output}[c, n]) \vee$
            $(n \in \mathit{fire} \wedge \mathit{input}[c, n] \wedge \mathit{output}[c, m]))$
    }
}
```

In the above definition, *input* and *output* are two helper predicate to test if node is an input, or is the output of a merger respectively. In our example, the relation *can_block* contains the tuples $\{(a, b), (b, a), (f, c), (c, g), (g, c), (m, g), (g, m), (m, b), (b, m)\}$. Note that the relation is symmetric except for the tuple (f, c) which is introduced by the LossySync $f \dashrightarrow c$.

Now we can define when a node is enabled to be fired: if it is not blocked itself, and cannot be blocked by any other blocked node in the circuit. This can be easily checked by getting a transitive closure of *can_block*:

```
pred $\mathit{enabled}\ [s : \mathit{State},\ n : \mathit{Node}]$ {
    $\neg\ \mathit{blocked}[s, n]$
    $\nexists\ m : \mathit{Node} \mid \mathit{blocked}[s, m] \wedge m \rightarrow n \in\ \hat{}(s.\mathit{can\_block})$
}
```

In the above example, the set of enabled nodes will be $\{e, f, c, g, m, d\}$.

Finally, the following fact imposes the maximal progress constraint on the traces generated:

$$\textbf{fact}\ \{\ \forall\ s : \mathit{State} - \mathit{last} \mid s.\mathit{fire} = \{n : \mathit{Node} \mid \mathit{enabled}[s, n]\}\ \}$$

We have implemented the above definitions and facts into the Relloy module, and have successfully analyzed circuits with context-sensitive behavior (like the one in the given example).

7 Conclusion and Future Work

We presented a method to model Reo circuits based on relational logic in Alloy. The resulting model preserves the original structure of the Reo circuit, and no complex translation effort is needed. This also makes the circuit description reasonably readable. Also, we have provided a library of different Reo channels as an Alloy module that can be reused when describing circuits. Our method handles basic channel types, compositional construction of more complex connectors, constraints on the environment, and circuits exposing context-sensitive behavior.

We can use Alloy Analyzer to verify properties on circuits. Properties are defined in terms of first-order predicates on the state of nodes and buffers in the execution traces of a circuit. As we can address states in our properties, along with 'next' and 'previous' operators and quantifiers, we can verify temporal properties on the circuit. Because Alloy Analyzer checks the properties on all possible traces, the properties are closely related to Linear Temporal Logic (LTL)[12] formulas [10]. More work is needed to precisely evaluate how expressive is this way to model temporal properties.

One can view our method as an implementation of Reo language in Alloy. But another useful viewpoint is to abstract away Alloy syntax, and view our work as a starting point to provide a formal semantics for Reo based on relational logic. More work is needed to formally define the semantics and compare it to the existing ones.

Our method currently ignores the actual values of data passed through the channels. Although many 'coordination' properties of a circuit can be expressed without explicitly modeling data values, adding this capability improves the expressiveness of the model in general.

Another issue to be addressed is scalability. Using Alloy Analyzer, it takes some time to analyze large connectors. An important observation here is that the description of the connector structure yields in only one instance. On the behavior side, once nondeterministic merge inputs are selected and the ready ports are defined, one can easily compute the *can_block* relation, its transitive closure, and finally the set of enabled nodes easily. In all these cases, we do not require solving SAT models. So, we can do parts of the computation in more efficient languages like Java (like construction of the connector instance from the description). The integration with Alloy can be done using Alloy API for Java. This may lead to a big improvement in the performance of analysis.

Currently, various tools on Reo have been implemented under the Eclipse platform [6]. Integrating our method with the tool set is another direction in which this work can be extended. This includes bi-directional transformation of graphical representation of a circuit to our textual format as both forms are necessary when working with models of different sizes.

Acknowledgment

We would like to thank Farhad Arbab for his helpful comments on our case study, leading to more detailed modeling of the environment.

References

1. Alloy Analyzer, `http://alloy.mit.edu`
2. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 1–38 (2004)
3. Arbab, F., Rutten, J.J.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
4. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.: Modeling component connectors in Reo by constraint automata. Science of Computer Programming 61, 75–113 (2006)
5. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: synchronisation and context dependency. In: Proceedings of FOCLASA 2005. ENTCS, vol. 154, pp. 101–119. Elsevier, Amsterdam (2006)
6. Eclipse Coordination Tools homepage, `http://homepages.cwi.nl/~koehler/ect`
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1994)
8. Jackson, D., Shlyakhter, I., Sridharan, M.: A micromodularity mechanism. In: Proceedings of Foundations of Software Engineering, pp. 62–73 (2001)
9. Jackson, D.: Alloy3.0 Reference manual (May 10, 2004), `http://alloy.mit.edu`
10. Jackson, D.: Software abstractions, logic, language, and analysis. MIT Press, Cambridge (2006)
11. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. In: Proceedings of FOCLASA 2006. ENTCS, vol. 175, pp. 19–37. Springer, Heidelberg (2007)
12. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York (1992)
13. Maude system, `http://maude.cs.uiuc.edu`
14. Mousavi, M.R., Sirjani, M., Arbab, F.: Formal semantics and analysis of component connectors in Reo. In: Proceedings of FOCLASA 2005. ENTCS, vol. 154, pp. 83–99. Elsevier, Amsterdam (2005)
15. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. Report SEN-R9834, CWI (December 1998)
16. Relloy Library, `http://ece.ut.ac.ir/msirjani/relloy.zip`
17. Talcott, C., Sirjani, M., Ren, S.: Comparing Three Coordination Models: Reo, ARC, and RRD. In: Proceedings of FOCLASA 2007 (to appear, 2007)