Integrating tools to reason about Reo circuits

Mariana Ferreira[1], **Bruno Lopes**[1]

[1]Instituto de Computação
Universidade Federal Fluminense (UFF)
Niterói-RJ -– Brazil

ferreira_mariana@id.uff.br, bruno@ic.uff.br

Abstract. *Critical systems are present in many applications and require high reliability. However, there are still challenges for the verification and certification of these systems. The graphical language Reo is based on coordination and model the communication of software components. A set of existing tools offers compilers and reasoners based on proof assistants and model checkers for Reo-specified systems. This paper proposes the integration of these compilers through an interface that allows building Reo circuits, converting the model to the compilers' input language, simplifying the use of logic tools and allowing the creation of new channels in addition to the canonical ones. The theory used, integrated tools, features of interface and some examples are presented.*

1. Introduction

Many software was developed for better reuse and componentization, synchronization and coordination such that these components became important. Regarding critical systems, automated tests only are not enough to guarantee their reliability and safety. Therefore, there is a need for methods to formalize and validate these systems, because they need a high level of reliability.

The Reo coordination language allows the construction of circuits that model these distributed systems. For graphical editing of a Reo circuit, there is a set of plugins called *The Eclipse Coordination Tools* [Arbab et al. 2008]. Although it allows the graphical representation of the circuit, it does not generate formal models, its focus is on the representation and analysis of the circuit. [Grilo et al. 2022] developed a tool to create Reo circuits and generate code to serve as input for the logical tooling and generate formal models. However, there are also limitations in [Grilo et al. 2022]: (a) the space for creating a model is fixed; (b) the interface on smaller screens is compromised; (c) names of existing nodes are sometimes changed when creating new ones, making it difficult to read; (d) there is no simple way to add new channels; and (e) to view the result generated by the logical tools, it is necessary to download a file, which makes it slower to use.

As an alternative to these tools, this project proposes the construction of an interface that integrates logical tools to formalize the circuit. This work consists of the construction of ReoXplore2, an interface capable of (a) creating Reo models through a graphical editor; (b) translating the visual model into the corresponding textual code and vice versa; (c) integrating the logical tooling, using Treo as input, to certify the built model; (d) allowing the addition of new channels just by importing and reusing the already created functions; in addition to (e) allowing the download of generated codes for future use and (f) ensuring the responsiveness of the interface and the canvas to allow the creation of larger and more complex circuits.

2. Reo

Reo [Arbab 2004] is a graphical language based on coordination with channels for the modeling and verification of systems. Each channel specifies a coordination pattern on the components connected by it. The emphasis is on the composition and behavior of channels, not the entities that communicate and cooperate through them. Without knowing the connected *software* components, the behavior of each Reo channel imposes a specific coordination pattern on the entities that read and scatter data across channels. So channels model the protocol between components and, therefore, Reo plays an important role in the integration of *software* components.

A Reo model can be defined as a graph of labeled edges, where edges are channels and nodes are components. Channels in Reo have two endpoints: the source point that disperses data to the connector and the destination point that accepts data from the connector's output. Each connector has a predefined behavior. Some examples are:

- $A \longrightarrow B$ Sync: accepts data at the source point if and only if it can disperse the data at the target point.
- $A \dashrightarrow B$ LossySync: In essence a Sync that always accepts data at the source point. If it cannot disperse the data at the target point, then the data is lost.
- $A \rightarrow\!\square\!\rightarrow B$ FIFO: accepts data at the source point and, if it cannot disperse it, the data is stored in the buffer until it can go to the target point.
- $A \rightsquigarrow B$ Filter: filters the data coming from the source point and only disperses it to the target point if the data passes the filter (a logical condition on the data); if not, the data is lost.
- $A \longrightarrow\!\!\blacktriangleright\!\!\longrightarrow B$ Transform: transforms the data coming from the source point according to a function and disperse its result at the target point.
- $A \rightarrowtail\!\!\leftarrowtail B$ SyncDrain: has two source points. When data is accepted at one source point, it is blocked until data arrives at the other source point, then both data are lost. If the data arrives at the same time, it is lost simultaneously.
- $A \rightarrowtail\!\!\parallel\!\!\leftarrowtail B$ AsyncDrain: similar to SyncDrain, but when data is accepted at one source point, it is lost in the connector immediately, without being blocked. If data is accepted in the other source point, it's also lost. If the data arrives at the same time, it is lost simultaneously.

To express more complex behaviors, it is possible to compose channels. In Figure 1, there is a circuit that represents a network of security cameras that send data to a central. Nodes 1, 3 and 4 represent the cameras, node 2 receives the data (the recordings), and node 5 is the central.

Nodes 1, 3 and 4 send the data to node 2 through the LossySync channel. This channel indicates that data can be lost along the way, which can occur due to a failure in internet connection, for example. Then, data successfully received by node 2 is sent to node 5 via the FIFO channel. The FIFO channel indicates that the data will wait in a queue and be sent one at a time to node 5.

In addition to the graphical language, there is also a textual syntax for Reo: Treo [Dokter and Arbab 2018]. Treo allows describing a Reo circuit in written form, in-

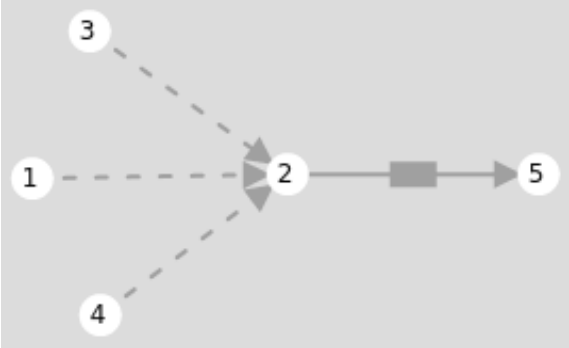


Figure 1. Reo circuit of a network of security cameras that send data to a central

dicating each connector and its respective nodes. This textual representation of the circuit is used to formalize the model afterward.

3. Constraint Automata

Constraint Automata [Baier et al. 2006] is the operating model for Reo proposed by the creators of Reo. The Reo language by itself does not allow the formalization of models, only their creation. So, using Constraint Automata as formal semantics for Reo enables proof assistants and model checkers to understand the Reo model to validate properties and certify the circuit.

Formally, *Constraint Automata* is defined as

Definition 1 (Constraint Automata)**.** *A Constraint Automaton (CA) is a tuple* $\mathcal{A} = (Q, \mathcal{N}ames, \rightarrow, Q_0)$ *where*

Q *is a finite set of states,*
$\mathcal{N}ames$ *is a finite set of port names,*
$\rightarrow : Q \times 2^{\mathcal{N}ames} \times DC \times 2^{Q}$ *is the transition relation that from a state Q, a set of ports and a data constraint (in propositional logic) leads to a set of states, and*
$Q_0 \subseteq Q$ *is a set of initial states.*

The circuit of Figure 1 is represented in Constraint Automata as follows: for the LossySync channel, there is a state with two transitions, the first transition in which the data passes from node A to node B, its condition is $d_A = d_B$, that is, the data from A must be equal to the data from B. The second transition is when the data is lost.

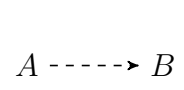
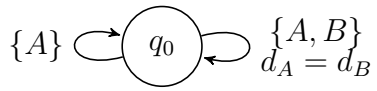
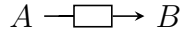
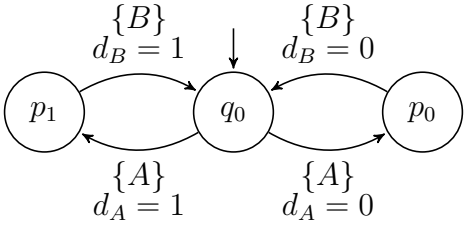
For the FIFO channel, there are three states and four transitions. The initial state is q0, if data 0 arrives from A in the buffer, the automaton performs transition to state p0, if data 0 arrives at node B, the automaton performs the transition back to state q0. If data 1 arrives from A in the buffer, the automaton performs transition to state p1, if data 1 arrives at node B, the automaton performs the transition back to state q0. This is for a binary alphabet, bigger alphabets need more states.

Table 1 presents the equivalences between the LossySync and FIFO channels of Reo and their respective Constraint Automatons.

4. Tools to reason about Reo

In this section, we present the logic tools integrated into this project, the CACoq and Reo2nuXmv compilers, and also describe the concepts of proof assistants and model checkers that are necessary to understand the purpose of the compilers.

Table 1. Equivalences between the LossySync and FIFO channels of Reo and their respective CA

Channel	Reo	*Constraint Automata*
LossySync		
FIFO		

4.1. CACoq

CACoq [Grilo et al. 2022] is a compiler responsible for generating Coq[1] code from the Reo circuit written in Treo. Coq is a proof assistant, which is a kind of software to assist in the development of formal proofs. They provide a formal language for writing mathematical definitions, algorithms, theorems and also automate routine aspects of proof construction [Pierce et al. 2018], however, it depends on human interaction for the non-automatic parts. Thus the demonstration becomes a joint task between humans and computers.

CACoq allows the formalization of Constraint Automata in Coq in order to obtain an environment for formal verification of a Reo circuit through the corresponding Constraint Automata. With the Coq certified model, it is possible to reason and proves properties about the Reo circuit. In addition, CACoq also allows you to extract the certified code to Haskell, a functional programming language. In this way, it is possible to guarantee the reliability of the built model and the behavior of the coordinated systems, thus obtaining the safety that the system behaves as specified.

4.2. Reo2nuXmv

Reo2nuXmv [Grilo et al. 2022] is a compiler responsible for generating nuXmv[2] code from the circuit on Treo. nuXmv is a model checker [Baier and Katoen 2008], which is a kind of software responsible for automatically checking the correctness of a formal system. It analyzes all possible states of the given model and verifies whether certain property is true or not.

As with CACoq, Reo2nuXmv also aims to formalize the Reo circuit through Constraint Automata. Reo2nuXmv models each canonical Reo channel for a Constraint Automata and, represented in a nuXmv `MODULE`, with its states and transitions. The final automaton, representing the entire circuit, is then built into a `MODULE` that uses these other created `MODULE`s, and each property check of the Reo model is done by executing this automaton and checking its states.

[1] `https://coq.inria.fr/`
[2] `https://nuxmv.fbk.eu/`

To show a part of the code generated by Reo2nuXmv, here is the formalization of the first LossySync channel from the Reo circuit of Figure 1. It's a `MODULE` whose value can vary between `NULL`, 0, or 1. Its current state begins with q0. The automaton transition happens when the current state is q0, and all ports are `NULL`, except port 1, then it goes to the next state that in LossySync is also q0.

```
1 --Channel from line 1 on the input file
2 MODULE lossySync1(time, ports)
3 VAR
4 var: real;
5 data: {NULL, 0,1};
6 cs: {q0};
7 TRANS
8 ((cs = q0 & ports.2[time] = NULL & ports.3[time] = NULL & ports.4[time] = NULL &  ports
      .1[time] != NULL) <-> next(cs) = q0);
```

There are two options for generating nuXmv code with Reo2nuXmv. In *compact* mode, performance tends to be better in terms of state-space and time to perform the verification, since the automaton undergoes an operation that minimizes it (to a certain extent). In *components* mode, the automaton does not go through any process other than the translation for each corresponding CA and its interactions, however, if an error occurs during the verification, it is possible to identify which component is causing the error, which facilitates and simplifies the correction process.

5. ReoXplore2

This work consists of the integration of these tools through an interface capable of: easily representing Reo models through a graphical editor; translating the visual model into the corresponding Treo code and vice versa; integrating the logical tools, using Treo as input, to certify the built model.

For the development, we used the Javascript programming language, the React.js [3] library to build the interface and the p5.js [4], a library for graphical programming, for the visual construction of the model. The source code is available at `https://github.com/frame-lab/ReoXplore2`. A lightweight version with reduced functionality, the creation of the graphical model and the translation to Treo, is also available at `https://frame-lab.github.io/ReoXplore2/`.

A flowchart of the functionalities of this work can be found in Figure 2. To model a Reo circuit, the user has the option of making a graphical input, building the circuit on the canvas, choosing the desired channels, and clicking on the canvas to create the nodes with these channels. Another option is for the user to perform the input by writing the Treo code in the textual area. In this case, the corresponding graphical model of the circuit is generated from the Treo code and displayed on the canvas. If the input is made through the graphical interface, then the circuit is modeled and the corresponding Treo code is automatically generated, being displayed in the textual area, where there is a button to download the file `treo.txt`.

After creating the Reo model, it is possible to change the positions of the nodes by pressing the 'd' key on the keyboard, this enables design mode, and the user can drag the

[3] `https://reactjs.org/`
[4] `https://p5js.org/`

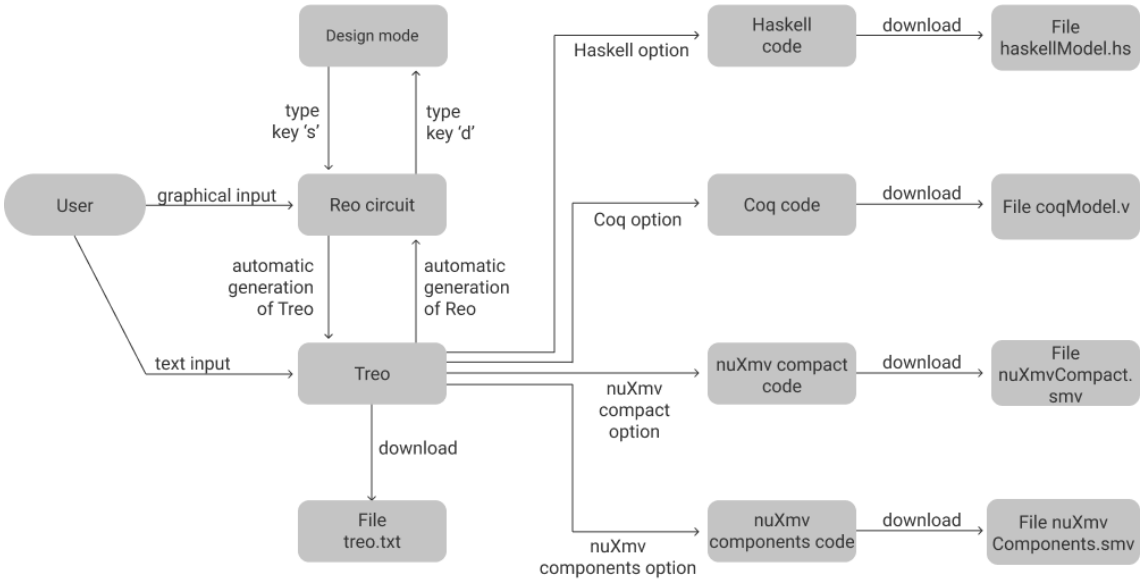


Figure 2. Flowchart of the functionalities

nodes with the mouse to change their position. Then, the key 's' saves the new positions and exits the design mode. Another important feature of graphic modeling is the resizing of the canvas, as it allows you to create circuits of larger sizes. The user can do it by dragging the mouse in the lower right corner of the canvas, which resizes the canvas both horizontally and vertically. And in order not to compromise usability, a scrollbar is added when the canvas' size becomes too big. Furthermore, the graphical construction of Reo channels was developed in a modularized way to allow the addition of new channels just by importing and reusing the already created functions: `line`, `triangle` and `center`.

Figure 3 shows a visual model of a Reo circuit on the left and the Treo automatically generated on the right. The first lines of the Treo are comments that indicate the positions of the nodes. The other lines are the code that demonstrates each channel that connects each pair of nodes. The visual model parser for Treo works as follows: With each update in the graph construction by the canvas, the *parser* goes through the list of objects of the created channels, and for each connector, it creates a *string* with the name of that channel followed by parentheses with the nodes that are being connected, like this: `channel.channelMode(startNode.label, endNode.label)` and concatenates with the result of the next channel. For each node of a channel, the *parser* creates a *string* to save the position of the nodes, like this: `# node.label (node.x, node.y)`. In the end, everything is concatenated into a single *string* and, thus, the visual model is transformed into Treo language text. This code is sent as input to the integrated compilers.

Once finished with the modeling, there are four options for formalizing the circuit: generating the model in Coq, generating the Haskell code, generating the nuXmv code in *components* mode or *compact* mode. When the user clicks one of these buttons, the interface makes a call and sends the Treo code to the local server. The server identifies the option selected by the user, writes the content of the Treo in a file called `input.txt` located in the corresponding compiler folder (previously installed) and runs the compiler.

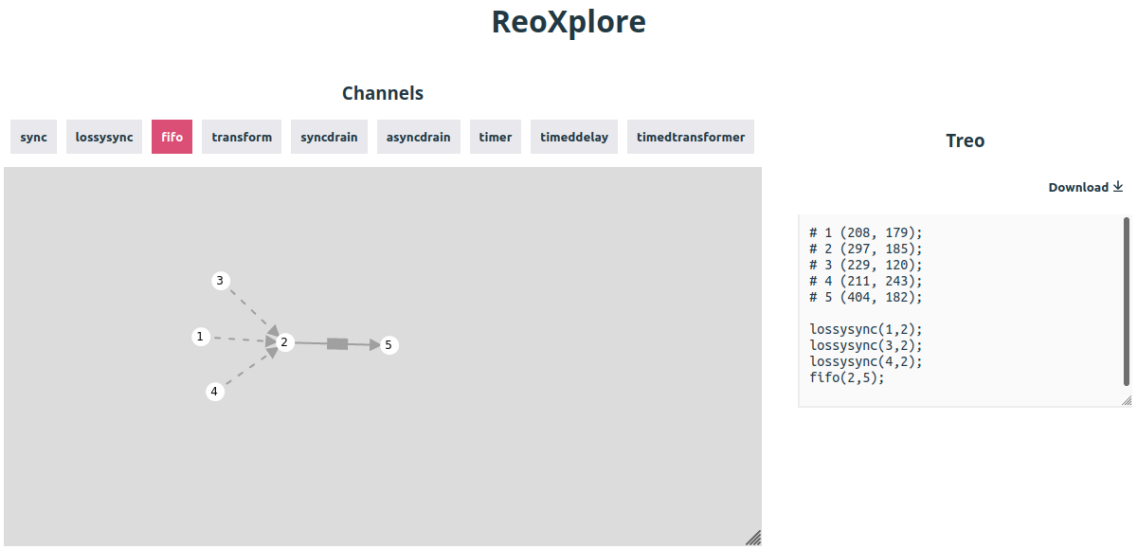


Figure 3. Treo generated from the visual model

The compiler, CACoq or Reo2nuXmv, reads the Treo and compiles it to the desired language, Coq, Haskell, nuXmv *components* or nuXmv *compact*, and then writes the result to an output file. The server reads that output file and responds the result back to the interface that displays it on the screen.

Figure 4 shows the interface, on the left are the options that the user can choose to compile the created Reo circuit, and on the right, the result code of CACoq compiler applied to the circuit of Figure 3. As with the Treo code, the user can download this code, which allows the later verification of properties or formal demonstrations of the model using proof assistants or model checkers.

Figure 4. Coq model generated from the Treo code using CACoq compiler

6. Conclusion and further work

In this work, we developed ReoXplore2, a tool to create Reo circuits and their integration with two others, CACoq and Reo2nuXmv. By using ReoXplore2, the user can create Reo circuits in a simple way, drawing a diagram that represents the circuit graph and translating to the corresponding Treo code automatically. The user can also make changes to the Treo code, which update the graph diagram instantly. Furthermore, the graphical construction of Reo channels was developed in a modularized way to allow the addition of new channels just by importing and reusing the already created functions. Finally, as ReoXplore2 integrates CACoq and Reo2nuXmv, it also works as an interface for using these two tools which allow the user to get the compiled code in Coq, nuXmv, or Haskell languages.

As further work, we can develop it in the following directions: (a) creation of a new functionality to integrate tdsReplLanguage [Torres and Lopes 2020], a high-level language used to represent timed data streams for Reo channels; (b) inclusion of the function of removing a node from the graph and its respective channels in the circuit construction visual module; and (c) extension of the lightweight version of the interface to provide all the functionality, for this it will be necessary to deploy the backend application and adapt the frontend, this will greatly facilitate its use, as no installation would be necessary, just a browser with internet to use the interface.

References

Arbab, F. (2004). Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(3):329–366.

Arbab, F., Koehler, C., Maraikar, Z., Moon, Y.-J., and Proença, J. (2008). Modeling, testing and executing reo connectors with the eclipse coordination tools. *Tool demo session at FACS*, 8.

Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT press.

Baier, C., Sirjani, M., Arbab, F., and Rutten, J. (2006). Modeling component connectors in reo by constraint automata. *Science of computer programming*, 61(2):75–113.

Dokter, K. and Arbab, F. (2018). Treo: Textual syntax for Reo connectors. *arXiv preprint arXiv:1806.09852*.

Grilo, E., Toledo, D., and Lopes, B. (2022). A logical framework to reason about Reo circuits. *Journal of Applied Logics - IFColog Journal of Logic and Their Applications*, 9:199–254.

Pierce, B., de Amorim, A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., and Yorgey, B. (2018). Software Foundations volume 1: Logical foundations.

Torres, M. A. and Lopes, B. (2020). Verificação de modelos com streams de dados sobre conectores reo. In *Anais do I Workshop Brasileiro de Lógica*, pages 9–16.