Automated Mapping of Reo Circuits to Constraint Automata

Fatemeh Ghassemi\textsuperscript{a,1} Samira Tasharofi\textsuperscript{a,2} Marjan Sirjani\textsuperscript{a,b,3}

\textsuperscript{a} Department of Electrical and Computer Engineering, University of Tehran, Tehran, Iran
\textsuperscript{b} School of Computer Science, IPM, Tehran, Iran

Abstract

A tool is developed in order to input Reo circuits and generate their corresponding constraint automata. The XML schemas for input and output are presented which establish a common interface for generating an integrated set of tools. Two heuristics in joining constraint automata are presented and experimental results are shown.

Keywords: Coordination languages, component-based systems, Reo, Constraint Automata, tool design

1 Introduction

Reo is introduced in [5,2], as a coordination language for modeling component-based systems. The components in a model are exogenously coordinated using Reo connectors. The simplest connectors in Reo are channels. Reo channels are connected through nodes and build up more complex connectors compositionally. So, a Reo circuit is made up of components, channels, and nodes as its elements.

A coalgebraic semantic model for Reo is presented in [6], in which connectors are relations on infinite timed data streams. Constraint automata are presented in [4] based on timed data streams, which capture operational

\textsuperscript{1} Email:f.ghasemi@ece.ut.ac.ir
\textsuperscript{2} Email:s.tasharofi@ece.ut.ac.ir
\textsuperscript{3} Email:msirjani@ut.ac.ir
semantics of Reo. Constraint automata provide compositional semantics for Reo where each Reo element is mapped to a constraint automaton and the model for the whole circuit is constructed using the join algorithm of constraint automata.

Efficient modeling of real-world systems in Reo and analyzing Reo circuits using constraint automata can only be done using automated tools. A set of integrated tools are needed in order to enter Reo circuits, map them to constraint automata, and then analyze the behavior of the model using constraint automata.

In this paper, we present a tool, RtC, for entering Reo circuits in XML format, and map the circuits to constraint automata. The input of the tool may also be multiple constraint automata to be joined. The architecture and the interfaces of the tool are designed in a way to provide extensibility, and heuristics are used in the main algorithms to provide efficiency. For developing the tool, a standard XML schema is defined for constraint automata, and the mapping of this XML schema and the existing standard XML schema for Reo [7] are established. The standard XML schemas for Reo circuits and for constraint automata provide a common interface between different tools supporting Reo. In implementing the join algorithm, heuristics are used in selecting two automata out of multiple automata to be joined, and also in obtaining the product of two automata.

Related work. Apart from the theoretical work on Reo and constraint automata, there are also some automated supports provided by tools. In [10,9] a tool is presented for joining constraint automata which is used to model and verify software architectural assemblies. The input and output of this tool is in text format and has a CSP-like style. The join algorithm only considers the transitions and the states play no role in obtaining the product. The RtC follows the work in [8], adding the Reo to constraint automata conversion functionality and also optimizing the join algorithm. The inefficiency of the join implementation in [8] which is only applicable for small examples shows the need for better algorithms and useful heuristics which are presented here.

In [11] another approach is used in order to automate reasoning about Reo models. An operational semantics is presented in SOS [12] rules, and it is translated to Maude [1] rewriting logic. Constraint automata are not used in this approach.

This paper is the first paper explaining a tool that automates the mapping of Reo to constraint automata, the standard interfaces for integrating the set of tools supporting Reo, and the algorithms and heuristics used in joining automata.
Structure of the paper. In Section 2, Reo and constraint automata are explained. Section 3 explains the tool using a simple example, and Section 4 describes the heuristics used in the tool and our experimental results. Section 5 contains our conclusions and future works.

2 Reo and Constraint Automata

Reo is an exogenous coordination language based on a calculus of channels [2]. Reo consists of components that are connected via connectors which coordinate their activities. Each component is a software implementation whose instances can be executed on physical and logical devices. Primitive connectors are channels which have two ends. There are two types of channel ends: source and sink. A source channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. Each channel end can be connected to at most one component instance at any given time. Complex connectors are constructed through composition of simpler ones by applying join operations. Channels are joined together in a node, so, a node is a construct which consists of a set of channel ends.

Channel Types. Reo has different types of channels. A channel may have a source and a sink end, two source ends, or two sink ends. Some basic types of channels, used in this paper, are the following:

- Sync: This channel has a source and a sink end, and no buffer. It accepts a data item through its source end iff it can simultaneously dispense it through its sink.
- LossySync: This channel is similar to the 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 (e.g., there is a take operation pending on its sink) the channel transfers the data item; otherwise the data item is lost.
- SyncDrain: This channel has two source ends. 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.
- FIFO-n: This channel has a source and a sink end, and a bounded buffer with capacity of $n$ data items. 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.
**Node Types.** The nodes in Reo, are categorized as:

- **Sink node:** All of the channel ends in the node are sink. A take operation from this node succeeds only if at least one of the coincident channel ends offers a data item. If more than one channel ends offer data, one is selected non-deterministically. A sink node, thus, acts as a (fair) nondeterministic *merger*.

- **Source node:** All of the channel ends in the node are source. A write operation on it succeeds only if all of the source ends in the node accept the data, in which case data is transparently written to every source end coincident on the node. Thus, a source node acts as a *replicator*.

- **Mixed node:** It contains both sink and source channel ends. It combines the behavior of a sink node and a source node in an atomic iteration of an endless loop: in every iteration, it 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. A data item is suitable for selection in an iteration only if it can be accepted by all source channel ends that coincide on the mixed node.

Reo provides operations that enable components to connect and perform I/O on sink and source nodes. Note that, a component cannot connect to, take from, or write to mixed nodes. At most one component can be connected to a (source or sink) node at a time. The I/O operations are performed through interface nodes between different components, which we call port nodes.

*Constraint automata* [4] are proposed as compositional semantics for Reo, based on timed data streams [6]. Each element of a timed data stream is a pair of time and a data item, where the time indicates when the data item is being input or output. A transition fires if it observes data item in a port of the component and according to the observed data, the automaton may change its state. A constraint automaton (over the data domain Data) is a tuple \( A = (Q, Names, \rightarrow, Q_0) \) where \( Q \) is a finite set of states, \( Names \) is a finite set of names, \( \rightarrow \) is a finite subset of \( Q \times 2^{Names} \times DC \times Q \), called the transition relation of \( A \), and \( Q_0 \subseteq Q \) is the set of initial states.

Constructing complex connectors out of simpler ones, is done by the join operation in Reo. Joining two nodes destroys both nodes and produces a new node on which all of their coincident channel ends coincide. Each channel in Reo, and the merger nodes are mapped to a constraint automaton. Some examples of this mapping are depicted in Figure 1 (taken from [4]). The single most important composition operator in Reo, join, amounts to a product of automata.

The product-automaton of the two constraint automata \( A_1 = (Q_1, Names_1, \rightarrow_1, Q_0_1) \) and \( A_2 = (Q_2, Names_2, \rightarrow_2, Q_0_2) \) is given by:

\[
A = (Q, Names, \rightarrow, Q_0)
\]

where \( Q = Q_1 \times Q_2 \), \( Names = Names_1 \times Names_2 \), and \( \rightarrow \) is defined as:

\[
\rightarrow = \{ (q_1, q_2) \mid (q_1 \rightarrow_1 q'_1, q_2) \in \rightarrow_1 \times \rightarrow_2 \} \cup \{ ((q_1, q_2), (q'_1, q'_2)) \mid q_1 \rightarrow_1 q'_1 \}
\]

and \( Q_0 = Q_0_1 \times Q_0_2 \).
Fig. 1. Constraint automata for some basic Reo channels, and merger node

\[ \rightarrow_1, Q_{0,1} \] and \( \mathcal{A}_2 = (Q_2, \mathcal{N}ames_2, \rightarrow_2, Q_{0,2}) \), is [4]:

\[ \mathcal{A}_1 \triangleleft \triangleleft \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}ames_1 \cup \mathcal{N}ames_2, \rightarrow, Q_{0,1} \times Q_{0,2}) \]

where \( \rightarrow \) is defined by the following rules:

\[
\begin{align*}
q_1 \xrightarrow{N_1 \cdot g_1} p_1, & \quad q_2 \xrightarrow{N_2 \cdot g_2} p_2, \quad N_1 \cap \mathcal{N}ames_2 = N_2 \cap \mathcal{N}ames_1 \\
\langle q_1, q_2 \rangle & \xrightarrow{N_1 \cup N_2 \cdot g_2} \langle p_1, p_2 \rangle
\end{align*}
\]

and

\[
\begin{align*}
q_1 \xrightarrow{N \cdot g_1} p_1, & \quad N \cap \mathcal{N}ames_2 = \emptyset \\
\langle q_1, q_2 \rangle & \xrightarrow{N \cdot g_2} \langle p_1, q_2 \rangle
\end{align*}
\]

and latter’s symmetric rule. □

The first rule is applied when in the automata there are two transitions which can be fired together. This happens only if there is no shared name in both automata which is present on one of the transitions but not present on the other one. In this case the transition in the resulting automaton has the union of the names on both transitions, the data constraint is the conjunction of the data constraints of the two transitions. The second rule is applied
when a transition in one automaton can be fired independently of the other automaton, which happens when the names on the transition are not included in the other automaton.

After a join operation, hiding in the result automaton can be done. Hiding abstracts the details of internal communication among channels in a Reo circuit, and shows the observable behavior of a Reo circuit. Model checking and analysis methods available for finite state automata and labeled transition systems can be adapted to the constraint automata [4].

3 RtC Tool

RtC is the tool for automating the translation of Reo to constraint automata. A Reo circuit may include different channel types as well as black box components. The corresponding constraint automata of a set of primitive channels are provided in RtC, and the behavior of the components must be specified as constraint automata as well. The input and output format for Reo channels and constraint automata are based on their corresponding standard XML schema [7]. RtC is implemented in Java and maps a Reo circuit to a constraint automaton, using the constraint automata join operation. Instead of inputting a Reo circuit, the user may input multiple constraint automata and perform the join and hide operations directly. Figure 2 shows the use case diagram of RtC tool.

![Fig. 2. The use case diagram of RtC tool](image)

The UML component diagram of the RtC tool is shown in Figure 3. The Command Interpreter package is used for interpreting user’s commands and their arguments and provides appropriate messages when errors are encountered. This interpreter uses the Automaton package to parse an automaton
in the XML format and to perform automata related operations as joining and hiding. The Command Interpreter also uses ReoCircuit package to parse a Reo circuit XML and convert it to its corresponding automaton. The Digest package is for parsing XML files according to their schemas.

Fig. 3. The component diagram of RtC tool

3.1 Reo and Constraint Automata in XML Format

In the following we shortly explain the XML format for both Reo and constraint automata, and then show an example.

XML Specification of Reo Circuits

A Reo circuit is defined in the XML format which is validated according to a standard Reo schema [7]. Each Reo circuit is defined by a circuit element in the XML file. A circuit element consists of two child elements named header and body elements. The header element contains a connector element which is used for defining the channel ends in the circuit that will be connected to external channel ends. The body element is a composite element whose constituent elements are channels, nodes and components. Each channel and node is specified by some predefined types in the standard XML schema of Reo. Each node is described by defining its coincident channel ends. Each channel is described by defining its two channel ends. A channel end is defined by assigning a unique identifier (number) to it. In the components element, we may have multiple connector instances. These are instances of predefined connectors existing in files.
**XML Specification of Constraint Automata**

A constraint automaton is also defined in the XML format which can be used as input or output of the tool. This XML file is validated by its corresponding schema provided in [8]. Each constraint automaton is defined by a `gxl` element which has the type of `graphType`. Each `graphType` is defined by two elements; `node` and `edge` elements for defining the states and transitions of an automaton respectively. The `node` element is a simple element that consists of one attribute `name`. The `edge` element is a composite element that consists of two elements, `signal` and `constraint`, for defining the names and data constraints of the transition respectively.

### 3.2 Example

In this section, we show a simple Reo circuit and its XML specification, and also the XML of its corresponding constraint automaton. We also briefly explain the mapping between two XML specifications. The Reo circuit is referred to as `Two-Sequencer` and is shown in Figure 4.

![Two-Sequencer Reo circuit](image)

This circuit receives data from channel ends, \( A \) and \( B \), simultaneously and orders them in the \( AB \) format. Thus, the accepting language of the corresponding automaton is \( (AB)^* \). This circuit has three port nodes, \( n1 \), \( n2 \), and \( n6 \), each including channel ends 1, 3 and 12 respectively. As noted above, these channel ends are defined in the header element of the XML definition, as shown in Figure 5. The internal nodes and channels are defined in the body element of the circuit, as shown in Figure 6.

The channel ends 1, 3, and 12 fire simultaneously, while there are write operations on channel ends 1 and 3, and a take operation on channel end 12. The data token on channel end 1, \( A \), is taken by channel end 12 and the data token of channel end 3, \( B \), is reserved in `FIFO1` (shown as c4 in Figure 4). When channel end 12 performs another take operation, the data token \( B \) is
<circuit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="./reo_circuit_schema_v1.0.xsd">
  <header>
    <title>Two Sequencer</title>
    <author/>
    <description/>
    <connector>
      <name>Sequence</name>
      <offers>
        <ce_ref ce_id_ref="1"/>
        <ce_ref ce_id_ref="3"/>
        <ce_ref ce_id_ref="12"/>
      </offers>
    </connector>
  </header>
  <body>
    <channels>
      <sync id="c1">
        <source ce_id="1" name="A"/>
        <sink ce_id="2"/>
      </sync>
      <sync id="c2">
        <source ce_id="3" name="B"/>
        <sink ce_id="4"/>
      </sync>
      <sync id="c3">
        <source ce_id="5"/>
        <sink ce_id="6"/>
      </sync>
      <fifo1 id="c4">
        <source ce_id="7"/>
        <sink ce_id="8"/>
      </fifo1>
      <syncdrain id="c5">
        <source ce_id="9"/>
        <sink ce_id="10"/>
      </syncdrain>
      <sync id="c6">
        <source ce_id="11"/>
        <sink ce_id="12" name="C"/>
      </sync>
    </channels>
    <nodes>
      <node id="n1" isHidden="false">
        <listCoincidentCEs>
          <ce_ref ce_id_ref="1"/>
        </listCoincidentCEs>
      </node>
      <node id="n2" isHidden="false">
        <listCoincidentCEs>
          <ce_ref ce_id_ref="3"/>
        </listCoincidentCEs>
      </node>
      <node id="n3" isHidden="true">
        <listCoincidentCEs>
          <ce_ref ce_id_ref="2"/>
          <ce_ref ce_id_ref="5"/>
          <ce_ref ce_id_ref="9"/>
        </listCoincidentCEs>
      </node>
      <node id="n4" isHidden="true">
        <listCoincidentCEs>
          <ce_ref ce_id_ref="4"/>
          <ce_ref ce_id_ref="7"/>
          <ce_ref ce_id_ref="10"/>
        </listCoincidentCEs>
      </node>
      <node id="n5" isHidden="true">
        <listCoincidentCEs>
          <ce_ref ce_id_ref="6"/>
          <ce_ref ce_id_ref="8"/>
          <ce_ref ce_id_ref="11"/>n
        </listCoincidentCEs>
      </node>
      <node id="n6" isHidden="false">
        <listCoincidentCEs>
          <ce_ref ce_id_ref="12"/>
        </listCoincidentCEs>
      </node>
    </nodes>
    <components/>
  </body>
</circuit>

Fig. 5. The XML specification for the Two-Sequencer Reo circuit: the header element

Fig. 6. The XML specification for the Two-Sequencer Reo circuit: the body element
taken from FIFO1 which makes it empty. The constraint automaton of this circuit has two states, each representing the empty and full status of FIFO1. This constraint automaton is shown in Figure 7. The XML definition of the constraint automaton for Two-Sequencer generated by the RtC tool is shown in Figure 8. The names on the transitions are the identifiers of the channel ends in the port nodes. These names describe which channel ends are fired simultaneously in the corresponding circuit.

Fig. 7. The constraint automaton resulted for the Two-Sequencer

\[
\begin{array}{c}
\text{s1} \\
\text{s0}
\end{array}
\]

1, 3, 12

\[
\begin{array}{c}
s1 \\
s0
\end{array}
\]

12

Fig. 8. The XML specification for the constraint automaton of the Two-Sequencer

4 Applied Algorithms

Different algorithms can be applied to perform the product operation between two automata (applying the rules of Section 2). One algorithm starts from production of the initial state(s) of the two automata and continues the operation considering only reachable states. Another algorithm ignores the states, and deal only with the transitions. In the latter algorithm we generate the entire state space resulting from the product of the two automata, and then traverse the resulting graph to remove unreachable states. These two algorithms are compared in Section 4.1.
Another issue that affects the performance of the product algorithm, is the order of the selection of two automata among several automata to be joined. Although the final results are the same using different orders, the intermediate results can be significantly different. So, in the mapping algorithm, when we are joining different channels, components, and merger nodes, we can use a heuristic for improving the performance. We apply a heuristic algorithm in the RtC tool which is explained in Section 4.2.

4.1 The Algorithm for the Product of Two Automata

In the first join algorithm only reachable states are generated and processed. The time complexity of this algorithm in the worst case is $O(n \times m)$ where $n$ and $m$ are the number of transitions in the automata. As the resulting transitions do not include the transitions of unreachable states, and each resulted transition is observed once during the algorithm, the time complexity never exceeds $n \times m$, and in many cases it is less than that. In the second algorithm the product of all transitions of the two automata are computed, then unreachable states are removed using the graph coloring algorithm. Therefore the time complexity of this algorithm is $O(n \times m + v)$ where $n$ and $m$ are the number of transitions of two automata and $v$ is the number of states in the result automaton. Thus, the first algorithm is more efficient than the second algorithm.

4.2 The Algorithm in Joining Several Automata (Sequencing Algorithm)

As noted above, one can use heuristics to improve the performance of joining several automata. In our heuristic algorithm in the RtC tool we consider the Reo elements that are mapped into constraint automata, including channels, merger nodes, and components. Note that a replicator node is not mapped into a constraint automaton. A Reo circuit is converted into an undirected graph of squares where each square represents an automaton, which is called square graph in this paper. Two squares are connected if their corresponding Reo objects are adjacent in the Reo circuit. Two Reo objects are adjacent if they are connected directly or by a replicator node.

The square graph for the Two-Sequencer is shown in Figure 9. Each channel and the merger node named $n5$ are presented by a square in the graph. The squares named $c1$ and $c5$ represent the automata resulted from channels named $c1$ and $c5$ respectively in Figure 4. These squares are connected in this graph because they are adjacent according to our definition via the replicator node named $n3$ in Figure 4.

The heuristic algorithm is based on joining two adjacent squares (au-
join adjacent squares causes their transitions to have at least one name in common, which results in fewer transitions according to the first product rule in Section 2. Selecting two automata, the product of whose transitions is minimum, results in less cost in each step, which improves the total cost of the products, although finding such two automata results in an extra cost in each step. In the following, we characterize the circuits where this overhead is tolerated by the improvement obtained in the total cost of products.

An alternate algorithm is the incremental algorithm, which starts from one square in the graph and joins other adjacent squares one by one. We can not use dynamic programming technique for finding the optimum sequence of products here, because we have a general graph instead of a line graph and also the squares (automata) can be joined in an arbitrary sequence.

The sequences of automata product for Two-Sequencer, following the heuristic and incremental algorithms, are shown in Figures 10 and 11. The sequences of automata product are performed from the inner most dotted squares toward outer ones. The number of transitions for each square (automaton) named $c_i$ or $n_i$ in the graph is represented by $C_i$ and $N_i$ respectively. When the incremental algorithm is applied, first $c_1$ and $c_3$ squares are joined by the cost of $C_1 \times C_3 = 1 \times 1 = 1$ and the resulted automaton has one transition. Then this automaton is joined to $n_5$ by the cost of $(1 \times N_5) = 1 \times 2 = 1$. Note that 1 is the number of transitions for the automaton resulted from $(c_1 \bowtie c_3)$. Thus, to calculate the total cost of products, the number of transitions for intermediate automata resulted in each step should be considered. The total cost of products applying the incremental algorithm on Two-Sequencer is $(((c_1 \bowtie c_3) \bowtie n_5) \bowtie c_6) \bowtie c_2) \bowtie c_4) = (C_1 \times C_3) + (1 \times N_5) + (2 \times C_6) + (2 \times C_5) + (2 \times C_2) + (2 \times C_4) = 1 + 2 + 2 + 2 + 4 = 13$. According to above explanation, the total cost of product applying the heuristic algorithm is $(((c_1 \bowtie c_3) \bowtie (c_5 \bowtie c_2)) \bowtie c_4) \bowtie (n_5 \bowtie c_6)) = 11$. The cost of join for heuristic algorithm is better than the incremental algorithm, for this example.
These algorithms are implemented and tested for several circuits taken from [7] (such as exclusive router-2, three sequencer with reset, cycler3) and two Reo circuits designed for real-world examples in [13]. In this experiment the execution time for both algorithms are compared as shown in the diagram in Figure 12. According to these empirical results, in the case of simple circuits these algorithms are almost equal but in the case of complex Reo circuits, our heuristic algorithm performs significantly better than the incremental algorithm. Finding two automata that the product of their number of transitions is minimum, causes an overhead in the total cost of heuristic algorithm, but in this experiment, this overhead is tolerated by the improvement obtained in the total cost of product.

In fact, this overhead is tolerated for a subset of circuits. We can assign to each square a weight according to the number of its transitions. Such a square
graph with weighted squares is called weighted square graph. We can categorize weighted square graphs into two categories named *homogeneous* and *non-homogeneous*; a *homogeneous* square graph is the one, where the average of weights for a square and its neighbors is almost the same for all squares. A weighted square graph which is not *homogeneous* is *non-homogeneous*. Roughly speaking in a *non-homogeneous* square graph, we can find regions that their squares are heavy or light and we call them heavy and light regions respectively. When the weighted square graph is *non-homogeneous*, applying heuristic algorithm will join all squares in a light region and then join this region to heavier squares or regions such that the cost overhead of search will be tolerated by the improvement obtained by the total cost of products. We applied heuristic and incremental algorithms to the different *non-homogeneous* graphs (Reo circuits), with sample patterns shown in Figure 13. The result is shown in Table 4.2.

![Graph with weighted squares](image)

**Fig. 13.** The different sample patterns for the *non-homogeneous* circuits
<table>
<thead>
<tr>
<th>Circuit type</th>
<th>Time for heuristic</th>
<th>Time for incremental</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type A</td>
<td>172</td>
<td>2740</td>
</tr>
<tr>
<td>Type B</td>
<td>325</td>
<td>460</td>
</tr>
<tr>
<td></td>
<td>8250</td>
<td>19250</td>
</tr>
<tr>
<td>Type C</td>
<td>203</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1820</td>
<td>10830</td>
</tr>
</tbody>
</table>

Table 1

The result of applying heuristic and incremental algorithms to the different patterns of non-homogeneous circuits. The time unit is millisecond.

Thus, our heuristic algorithm performs significantly better than incremental algorithm for non-homogeneous circuits.

Using heuristic algorithm, may cause the join operation to be performed on the automata in separate parts of the graph. This helps us in joining very large circuits which is another advantage of our heuristic algorithm. In case of memory shortage, we can move heavier squares (automata) to a secondary memory as they are not required in earlier iterations of heuristic algorithm.

5 Conclusion and Future Work

In this paper, we explained the RtC tool for modeling Reo, and mapping Reo circuits to constraint automata in order to analyze their behavior. We explained the architecture and the format of the interfaces of the tool, and described the heuristics used in the main algorithms.

A graphical user interface will soon be added to RtC. This interface, enables the user to input Reo circuits or constraint automata in graphical format. RtC then generates the XML specifications of its input. A graphical output is also generated by RtC through the XML file of the output constraint automata. The next version of RtC applies a memory management policy to handle very large Reo circuits and automata.

We are currently examining other heuristics that can be used in the join algorithm. The parameters that are considered are the number of neighbors of each square in the square graph, and the number of transitions that are merged by a product of two constraint automata. We can improve the heuristic algorithm by using parallel techniques in implementation. We are also investigating the set of well-formed constraint automata which can be derived from Reo circuits.

In RtC, data constraints are considered during the join operation, but are
not evaluated. The evaluation of data constraints to false removes a transition, and thus, causes a reduction in space. In the next version of RtC, evaluation of data constraints will be implemented. Extensions of constraint automata have been proposed [3] which will be supported in future versions of RtC.

Acknowledgement

We are grateful to Mohammad Taghi Hajiaghayi for his useful comments on analyzing the algorithms discussed in this paper.

References


