Efficient Join Order for Constraint Automata through LLM-Generated Heuristics

Ali Mehrani*[1], Fatemeh Ghassemi†[1], Marjan Sirjani‡[2], and Farhad Arbab§[3]

[1]University of Tehran, Tehran, Iran
[2]Mälardalen University, Västerås, Sweden
[3]CWI, Amsterdam, The Netherlands

Abstract

Reo is an exogenous coordination language designed for component-based systems based on channel-based connectors. Constraint automata is defined by Christel Baier et al. as the compositional operational semantics of Reo. Semantics of a Reo circuit is computed by joining the constraint automata of the connector elements. This computation can be costly when dealing with large connectors, making an improvement necessary. Improving this operation involves either improving the join algorithm or selecting a joining order that minimizes intermediate automata. While alternative algorithms for joining constraint automata have been proposed, identifying an efficient joining order remains a challenge. This paper proposes a heuristic-based approach for finding an efficient order of joining constraint automata. By feeding OpenAI's ChatGPT with data on the join algorithm and the structure of constraint automata, we ask it to generate diverse heuristics to identify the most efficient joining order and employ its suggestions. Our results demonstrate the impact of join order on the operation's performance. We analyze these results to identify the best heuristic for each set of CAs based on their characteristics. This highlights the potential of LLM-driven approaches in assisting the development of efficient solutions for computational tasks.

Keywords: Constraint Automata, Reo, Coordination languages, Component-based systems, Large Language Models

Dedication. This paper is dedicated to Christel Baier in celebration of her technical and leadership contributions to advancing the field of formal verification in computer science.

1 Introduction

Modern distributed systems are widely used as solutions for solving problems when multiple computers have to work together to achieve their goal. These systems partition tasks among themselves and solve them separately, while also communicating with each other, sharing results, and processing tasks in parallel. Examples of such systems include file-sharing systems, large-scale databases, and service-oriented applications, which are widely adopted solutions for such scenarios. However, using and managing such systems has its own challenges, such as resilience to hardware, software, and network failures. One of the most important among them is managing coordination and concurrency due to the use of different communication protocols. Therefore, it's essential to understand the behavior of these component-based systems.

*ali.mehrani@ut.ac.ir
†fghassemi@ut.ac.ir
‡marjan.sirjani@mdu.se
§farhad@cwi.nl

A good practice for countering this challenge is to separate the system's computation logic from its coordination logic. This allows for monitoring and modeling the coordination mechanism without addressing each node's computational complexity. To achieve this goal, coordination languages have emerged to model the coordination of component-based systems, including distributed and embedded systems. Reo is one of the most well-known coordination languages, introduced by Arbab [1] in 2001. The coordination among the components is realized by Reo connectors made of the built-in connectors, called channels, connected to each other through nodes. The data is later passed through the nodes and channels in this circuit. Reo can be used to model and analyze the interaction and communication of different components in a system, especially service-oriented applications. For example, in a work by Tasharofi et al. [10], the interaction of different web services has been modeled using Reo.

The operational semantics of Reo circuits are defined by constraint automata (CAs), which capture the system's data flow. The constraint automaton of a Reo circuit is achieved by joining the constraint automata of its constituent elements, i.e., channels and nodes. There are some tools developed that convert a Reo connector to its constraint automata for verification purposes [4, 7, 8]. The result of the join operation is not sensitive to the join order, which means that if there were more than two automata to be joined, no matter which two are joined first, the result will always be the same. Different join orders lead to different intermediate automata, which might vary in size. The bigger the intermediate states get, the less efficient the entire operation will be. This is very similar to relational databases, where joining different tables does not affect the result, but can highly affect the operation's performance [2, 5].

An inefficient implementation of the join algorithm can significantly reduce performance when dealing with large CAs, as mentioned in [4]. Therefore, it is crucial to improve the efficiency of the join operation. This can be achieved by improving the join algorithm while also identifying an efficient joining sequence that minimizes the size of intermediate automata, reducing the operation's computational load. While alternative algorithms for the former concern have been proposed [6, 9], identifying an efficient join order remains a challenge.

Related Work. Automated conversion of Reo circuits to constraint automata has been introduced in some tools that employed different approaches for enhancing the operation's efficiency [5, 6, 9]. The tool introduced in [5] converts Reo circuits into their corresponding CA by employing a heuristic based on the number of transitions of CAs of adjacent elements to find the order of joining CAs. The employed heuristic examines all pairs of automata that share at least one transition label in common and selects the pair with the smallest transition product for joining. An alternative algorithm for constraint automata join was proposed by Pourvatan and Rouhy [9], being more efficient than the previously proposed algorithms by a constant factor. They also introduced a greedy algorithm for finding the selection order of CAs based on their shared transition labels, aiming to minimize the number of transitions in the intermediate CAs. Their proposed algorithm picks two automata with the largest set of common names in each iteration. An approach for grand composition (production) of constraint automata was introduced by Jongmans et al. [6]. The grand compositions are computed *state-by-state* instead of *iteratively*. Thus, only the reachable states of the final CA are computed, avoiding the unnecessary computation of intermediately reachable eventually leading to unreachable states.

In this paper, we present a heuristic-based approach for conducting the join operation on multiple constraint automata in such an order that the intermediate CAs become the smallest possible in size, which lowers the total operation's space consumption and

boosts its performance. We provided [2] and [5] as context to OpenAI's ChatGPT, then prompted it to generate heuristics to be applied in the join operation. The heuristics are mainly compared by the operation's total computation time and the size of the intermediately generated CAs.

2 Preliminaries

In this section, we briefly introduce Reo connectors, Constraint Automata, and the join operation on Constraint Automata.

2.1 Reo

Reo is an exogenous coordination language used for the compositional reasoning and construction of component connectors, which allows to separate coordination from the behaviors of components. The Reo Connectors consist of channels and nodes that connect components. Each channel can be connected to at most a single component instance at any given time. A channel consists of two ends and a constraint that relates the timing and content of the data flow. Reo has two types of channel ends: *source* and *sink*, where the data is entered through the source end and distributed through the sink channel end. A node is a logical construct that consists of a set of channel ends. There are three types of nodes based on their containing channel ends: source, mixed, and sink. Data flows from source nodes to sink nodes [1, 2]. All channel ends in a source node are of source type. A write operation on this node is only possible when the data is accepted by all the source channel ends. Therefore, a source node acts as a *replicator*. A take operation from a mixed or sink node is possible if at least one of its sink channel ends offers a data item. If data is offered by more than one sink channel end, one is selected non-deterministically. Therefore, a sink (or mixed node with more than one sink channel end) node acts as a non-deterministic *merger*. Reo enables gluing channels of arbitrary types to build component connectors [1, 3].

Reo provides a set of primitive channels that can be joined together to form a node in a Reo circuit, where channel ends coincide. Figure 1 shows basic Reo channels *Sync* (1a), *LossySync* (1b), *SyncDrain* (1c), *FIFO-1* (1d), and the merger node (mixed node) (1e) along with their corresponding constraint automata.

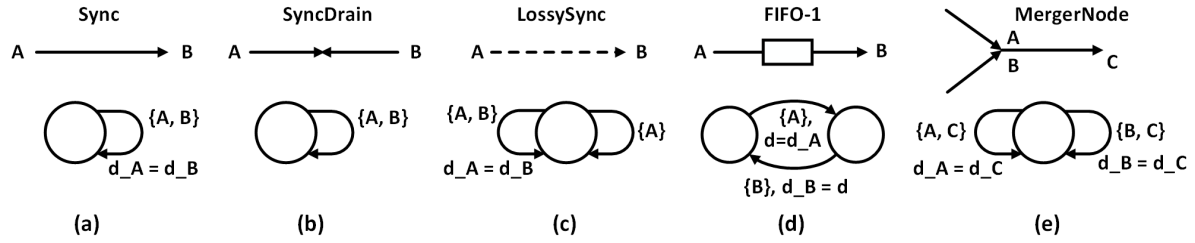


Figure 1: Graphical representation for basic Reo channels Sync, SyncDrain, LossySync, FIFO-1, and the merger node along with their corresponding constraint automata

2.2 Constraint Automata

Constraint automata (CAs) was introduced in [2] to define the operational semantics of Reo circuits. In these automata, states represent the possible configurations, such as

the contents of FIFO channels or other buffered channels, and transitions represent the possible data flow where their labels can be viewed as *sets of I/O-operations* that will be performed *in parallel* on channel ends.

Figure 1a shows a *Sync* channel with the source channel end A and the sink channel end B, along with its corresponding constraint automaton. The constraint automaton for the *Sync* channel has only one state, as this channel lacks a buffer or any other medium to store messages. The single transition in this CA represents the flow of data through the channel from A to B, while the data constraint ensures that the data item received at the source is identical to the one delivered at the sink. Similarly, Figure 1d shows a *FIFO-1* channel alongside its constraint automaton, with the source end A and sink end B. Assuming that the data domain consists only of the data item d, the initial state of its CA represents an empty buffer, while the second state corresponds to the buffer filled with d. The transition from the initial state to the second state represents the data item d entering the buffer, while the reverse transition represents d leaving the buffer, returning it to an empty state.

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

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

We write $q \xrightarrow{N,g} p$ *instead of* $(q, N, g, p) \in \rightarrow$*. We call* N *the name-set and* g *the guard of the transition. For every transition*

$$q \xrightarrow{N,g} p$$

we require that: (1) $N \neq \emptyset$ *and (2)* $g \in DC(N, Data)$ *is data constraint over* N *and Data.* $\mathcal{A}$ *is called finite iff* Q*,* $\rightarrow$*, and the underlying data domain Data are finite.* □

The CA of a Reo circuit can be obtained by joining the CAs of its constituent elements, i.e., channels and nodes. This procedure combines each two nodes into a single node and creates a new CA from the two input CAs.The join algorithm for automata production is defined as follows:

Definition 2 ***(Product automaton).*** *The product automaton of the two constraint automata* $\mathcal{A}_1 = (Q_1, \mathcal{N}ames_1, \rightarrow_1, Q_{0,1})$ *and* $\mathcal{A}_2 = (Q_2, \mathcal{N}ames_2, \rightarrow_2, Q_{0,2})$ *is:*

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

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

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

and

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

and the latter's symmetric rule. □

In this definition, $\mathcal{N}$ of *names* is a finite set, e.g., $\mathcal{N} = \{A_1, \ldots, A_n\}$ where A_i stands for the i-th input/output port of a connector or component [2].

3 Heuristics for Joining Constraint Automata

To generate heuristics for joining CAs, we provide ChatGPT with data on the join algorithm and the structure of constraint automata, then ask it to create different heuristics for us. GPT suggested six heuristics. Among them, the transition-based heuristic, called *Min Transitions Heuristic* in Section 3.1, was similar to the heuristic proposed in [5], and the heuristic based on shared transition labels, called *Max Connectivity Heuristic* in Section 3.1, was used in both [5] and [9]. We applied these heuristics in various scenarios, each generating a specific sequence of CAs to be joined. Regardless of the heuristic used, the result of the join operation remains the same as expected. However, the applied heuristic may alter the join order and intermediate results, causing an improvement or, in rare cases, a decline in the operation's performance. The heuristics generally aim to avoid the generation of large intermediate CAs, also known as *intermediate compounds* [5, 6, 9].

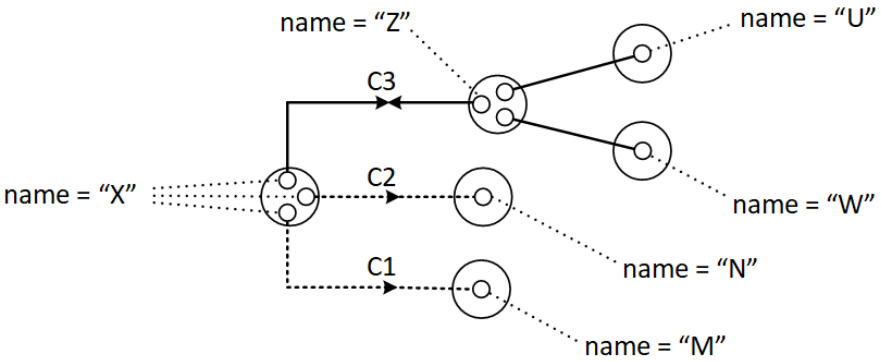


Figure 2: The running example Reo circuit

We take advantage of a running example to show the applied joining order resulting from each heuristic. Figure 2 shows the Reo circuit for our running example containing two *LossySync* channels named C1 and C2, a *SyncDrain* channel named C3, a merger node, and a replicator node. Figure 3 demonstrates the corresponding CAs that should be joined, and their final product; two CAs for *LossySync*s along with the CA for *SyncDrain*, a CA for the merger node.

We convert our Reo circuit to an undirected graph of squares shown in Figure 4. In this graph, squares are connected by lines if their corresponding Reo elements are adjacent to each other in the Reo circuit. Elements in Reo are adjacent if they are connected directly or via a replicator node. Since replicator nodes have no constraint automata, the CA of the Reo circuit in Figure 2, is achieved by joining the CAs of channels C1, C2, C3, and the merger node, which are named CA1 to CA4. According to this definition, CA1, CA2, and CA3 are adjacent in our square graph because they are adjacent in Figure 2 via the replicator node, and CA3 and CA4 are adjacent in our square graph because they are connected directly in Figure 2. We later use this square graph to visualize different joining sequences of constraint automata determined by our heuristics.

3.1 LLM-generated Heuristics

We introduce the LLM-generated heuristics and explain their working procedures. Later, in Section 4, we elaborate on the effect of these heuristics on computation time and memory consumption, leveraging a set of experiments. If we do not use any heuristic for the join operation, the default join order would be $(((CA1 \bowtie CA2) \bowtie CA3) \bowtie CA4)$,

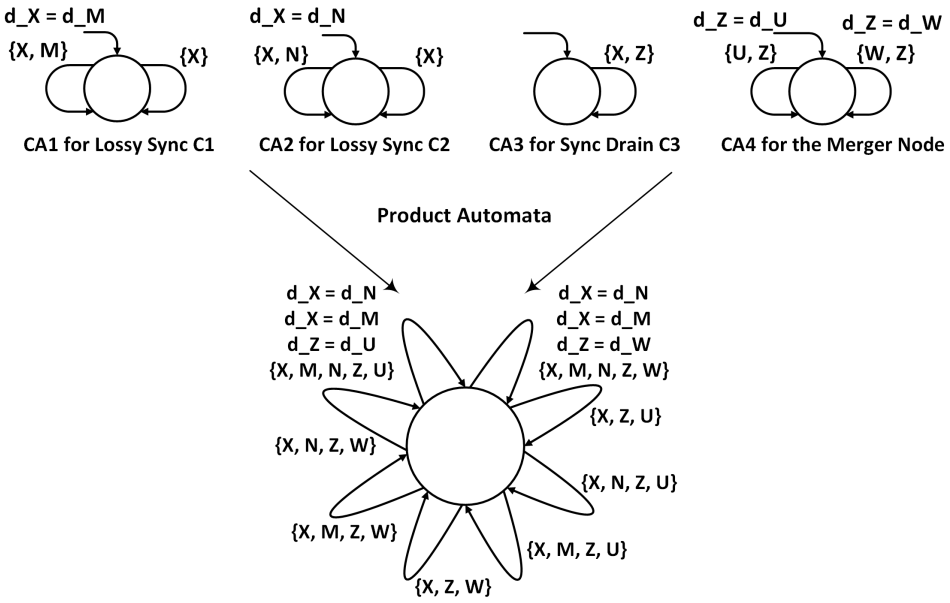


Figure 3: The CAs of the Reo elements in the running example in Fig. 2 and the CA generated by their join.

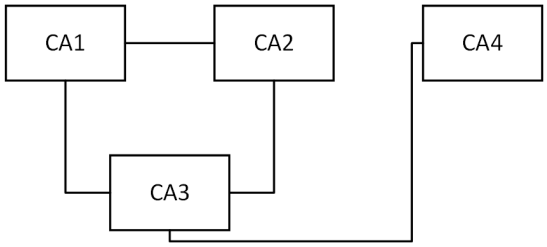
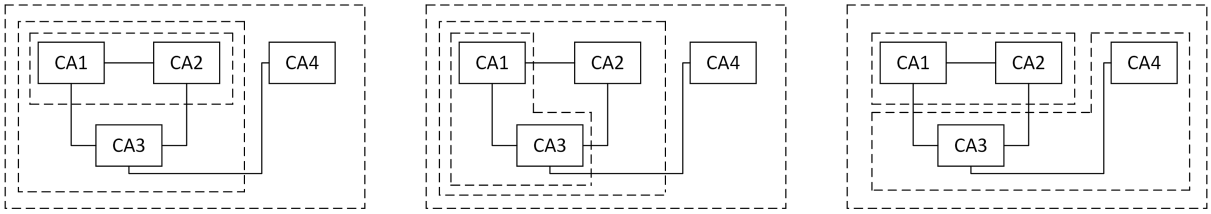


Figure 4: The original square graph for the proposed running example, where each square represents a constraint automaton. CAs are connected by lines if their corresponding Reo elements are adjacent in the Reo circuit shown in Figure 2.

which is shown in Figure 5a. We also assume that, when the heuristic values are equal, the two CAs with the lowest IDs are picked for the join, the same as the default approach.

Min Transitions Heuristic. This heuristic is similar to the one proposed in [5]. However, unlike that approach, it selects the two CAs with the fewest transitions among all available CAs and computes their product, instead of restricting the selection to adjacent ones (which share at least one transition label). The number of transitions in all CAs, including intermediate ones, is used as their heuristic score. It considers the number of transitions in a CA as an indicator of its size, prioritizing CAs with fewer transitions to keep intermediate CAs small. Figure 5b shows the joining sequence for our proposed example, generated using the *minimum-transitions* heuristic. It first selects CA3, as it has the fewest transitions (only 1). Among the remaining CAs, which all have 2 transitions, it picks CA1 (the one with the lowest ID) to be joined with CA3. The resulting CA from this join has 1 state and 2 transitions. The remaining CAs—(CA1 $\bowtie$ CA3), CA2, and CA4—all have the same number of transitions, and the heuristic selects (CA1 $\bowtie$ CA3) and CA2 for the next join iteration. The final join sequence generated by this heuristic is (((CA1 $\bowtie$ CA3) $\bowtie$ CA2) $\bowtie$ CA4).

Min States Heuristic. This LLM-suggested heuristic is similar to the previous heuristic, except that it considers the automaton's number of states as its score value, selecting



(a) The default join order

(b) Join order (((CA1 ⋈ CA3) ⋈ CA2) ⋈ CA4) selected by the *Min transitions* and *Transition Density* heuristics

(c) Join order ((CA1 ⋈ CA2) ⋈ (CA3 ⋈ CA4)) selected by the *Transition Disparity* heuristic

Figure 5: The joining sequences generated by the LLM-suggested heuristics along with the default sequence. The dashed line between CAs denotes their join.

two CAs with the fewest states in each step. This criterion is also one of the main indicators of an automaton's size, making it useful for selecting the smallest CAs first during the join operation. For our proposed example, since the number of states in all CAs, including the intermediately generated ones, is equal to 1, this heuristic generates the same joining sequence as the default approach shown in Figure 5a.

Transition Density Heuristic. This heuristic is calculated by dividing the total number of transitions by the total number of states in the automaton. It selects CAs with fewer transitions per state. In our running example, CA3 has 1 transition and 1 state, making it the CA with the lowest transition density value of 1, compared to other CAs, each of which has 2 transitions, 1 state, and a transition density value of 2. As a result, CA3 is selected first. Since the transition density values of the remaining CAs are the same, CA1, having the lowest ID, is chosen to be joined with CA3. The CA generated from this join, CA1 ⋈ CA3, has 2 transitions and 1 state, the same as the other remaining CAs. Thus, CA2 is selected to be joined with CA1 ⋈ CA3. The final joining sequence is (((CA1 ⋈ CA3) ⋈ CA2) ⋈ CA4) as shown in Figure 5b.

Transitions Disparity Heuristic. The idea behind this heuristic is to avoid joining automata with significantly different transition counts. It selects a pair of CAs with the closest number of transitions and joins them, rather than joining CAs with a large difference in transition counts. Using this heuristic, CAs with the smallest difference in their number of transitions are joined first. In our proposed example, CA1, CA2, and CA4 each have 2 transitions, while CA3 has 1. Any pair among CA1, CA2, and CA4 has a transition disparity value of 0, the minimum possible for this heuristic, whereas pairing CA3 with any other CA results in a disparity value of 1. Among the possible pairs within CA1, CA2, and CA4, CA1 and CA2 are selected first, producing an automaton with 1 state and 4 transitions. After this join, CA1 ⋈ CA2 has 4 transitions, while CA3 has 1, and CA4 has 2. The heuristic then selects CA3 and CA4 to be joined, as their transition disparity value (1) is the lowest among all remaining pairs. The final joining sequence is ((CA1 ⋈ CA2) ⋈ (CA3 ⋈ CA4)), as shown in Figure 5c.

States Disparity Heuristic. This heuristic works similarly to the previous one, except that it measures the difference in the number of states between automata. Thus, CAs with the smallest difference in their number of states are selected first for joining. In our running example, since all CAs—and any intermediate automata generated during the process—have exactly one state, the joining sequence remains the same as the default approach shown in Figure 5a.

Max Connectivity Heuristic. This heuristic is the same as the selection criterion proposed in [9]. According to Definition 2, joining two CAs that have more transition labels in common generally results in a smaller CA compared to joining two CAs with the same number of transitions and states but fewer shared transition labels. This corresponds to the concept of connectivity in Reo circuits, in which CAs that have more transition labels in common are more connected in the original Reo circuit. Thus, this heuristic prioritizes joining automata that are highly connected in the original Reo circuit. In our proposed example, CA1, CA2, and CA3 share one transition label (X), while CA3 and CA4 share another label (Z). Since CA1 and CA2 have the lowest IDs, they are selected first for joining. The resulting automaton, CA1 $\bowtie$ CA2, also shares one transition label with CA3 and is then joined with it. The joining sequence is the same as the default joining sequence in Figure 5a.

Each of these heuristics might alter the operation's performance based on the features of the input automata. We demonstrate the impact of different heuristics on the operation's efficiency through a set of test cases (experiments).

4 Evaluation Results

In this section, we evaluate the impact of the heuristics by analyzing the execution results of the join operation and their impact on performance. For this analysis, we consider two main factors: (1) the computation time of the operation and (2) memory usage, measured by the size of the intermediate automata (where the number of transitions in a CA indicates its size). We explain our developed framework for evaluating the operation's performance, later showing the operation's results along with a discussion. Implementation details can be accessed at: https://github.com/UT-ECE-FormalMethods/RtC.

4.1 Evaluation Framework

Our evaluation framework for joining CAs is developed in Java and assigns a heuristic score to each of them. The algorithm iteratively picks two CAs with the lowest or highest scores, depending on the selected heuristic, generates the product automaton, calculates its new heuristic score, and pushes it back to the list of CAs, ordered by their heuristic scores. The algorithm continues until one CA, which is the final result is remained in the list. We evaluate each heuristic across different test cases, each containing a set of CAs to be joined, which we elaborate on further in Section 4.2. During this process, we ensure that no optimization, such as caching, occurs and that the operation only uses heuristics to improve its performance. We remark that unreachable states are removed only from the final CA, as a state that is unreachable in an intermediate CA may become reachable in the next joins. Removing these states earlier could lead to an incorrect result.

Our framework is mainly divided into two parts, (1) the *SingleJoin*, which joins two constraint automata using the algorithm discussed in Section 2.2, and (2) the *MultiJoin* which takes a list of CAs and joins them in an order determined by the selected heuristic. The latter maintains a list of the CAs, picks them based on their heuristic values, and measures the time taken to join in the determined sequence. Regardless of the time complexity of the *SingleJoin*, a different joining sequence will affect the operation's total performance. In our framework, constraint automata are represented in JSON format. During this process, the ID of the resulting automaton in its corresponding JSON notation denotes the joining sequence used for its generation. For example, if the ID of the final CA is (CA3 $\bowtie$ ((CA1 $\bowtie$ CA2) $\bowtie$ CA4)), it denotes that, to construct the final CA, CA1

and CA2 were first joined, then the intermediately generated CA was joined with CA4, and finally, the resulting CA was joined with CA3 to create the final CA. The framework will measure the total execution time, along with the number of transitions and states of intermediate CAs, to compare different joining sequences determined by the proposed heuristics. Each joining sequence determined by a heuristic is executed 10 times, and the average execution time is considered the total time spent on the operation.

4.2 Evaluation Results

In this part, we discuss the evaluation results of the heuristics on our test cases. We compare the efficiency of the heuristics based on two criteria: computation time and the size of intermediate automata (where the number of transitions in a CA indicates its size), which indicates memory usage. We evaluate the heuristics on a total of 20 test cases, where each test case contains a set of CAs to be joined. For better visualization of the operation's performance, we divide these test cases into four categories: Small, Medium, Large, and X-Large based on the average execution time for the join operation. Each category contains 5 test cases. The execution time for small test cases ranges from a few milliseconds to slightly more than a second, while medium test cases typically take between 1 and 25 seconds. Large test cases range from tens of seconds to several minutes, and the X-Large test cases have execution times ranging from a few minutes to up to 4 hours. Table 1 provides more information about our experiments, where we introduce each experiment along with its characteristics, including the average number of states and transitions of the CAs, the average size of their transition label set (Names), the maximum number of transition labels they share, and the total number of CAs to be joined.

Table 1: Characteristics of the fifteen test cases used for evaluation in our framework. Each test case contains a set of CAs to be joined. Test cases are classified into four categories of Small, Medium, Large, and X-Large based on the average time taken for the join operation on the CAs in each test case.

		Test Case S-1	**Test Case S-2**	**Test Case S-3**	**Test Case S-4**	**Test Case S-5**
Small (S) **Join Time ≤ 1 sec**	Total CAs to be joined	3	3	3	4	4
	Avg # of states	3	4	7	5	5.5
	Avg # of transitions	3.3	5.3	8	6	7
	Avg # of CA names	3	4	3.3	4.75	4
	Max # of shared names	1	2	3	4	3.75
		Test Case M-1	**Test Case M-2**	**Test Case M-3**	**Test Case M-4**	**Test Case M-5**
Medium (M) **1 sec≤ Join Time ≤25 sec**	Total CAs to be joined	3	4	5	4	4
	Avg # of states	9	8.5	5.6	7.25	6.5
	Avg # of transitions	15.6	18.75	6.6	11.75	9.25
	Avg # of CA names	5	8	4	5.75	4
	Max # of shared names	4	8	5	2	2
		Test Case L-1	**Test Case L-2**	**Test Case L-3**	**Test Case L-4**	**Test Case L-5**
Large (L) **15 sec≤ Join Time ≤37 min**	Total CAs to be joined	5	4	4	4	4
	Avg # of states	7	10	9.5	10.5	13.25
	Avg # of transitions	12	21.25	12.5	16.25	23.25
	Avg # of CA names	4.6	10	5.5	9	9.5
	Max # of shared names	6	8	3	6	8
		Test Case XL-1	**Test Case XL-2**	**Test Case XL-3**	**Test Case XL-4**	**Test Case XL-5**
X-Large (XL) **3 min≤ Join Time ≤4 hrs**	Total CAs to be joined	6	7	8	9	10
	Avg # of states	5.66	4.85	3.5	3	2.5
	Avg # of transitions	9.16	9.42	5.625	3.77	3.7
	Avg # of CA names	3.83	4.57	4.625	3	2.8
	Max # of shared names	4	6	6	2	4

The execution times were measured on a system equipped with an Intel Core i7-9750H CPU and 16 GB of memory. We expect the join operation to generally have a better performance when using heuristics compared to the default approach.

Figures 6, 7, 8, and 9 show the computation time and the average number of intermediate CAs transitions for small, medium, large, and X-large test cases, respectively, where heuristics are compared with the default approach in both of these criteria. To

enhance the visualization of our plot, a small jitter value is added to the plotted values to prevent the lines from overlapping.

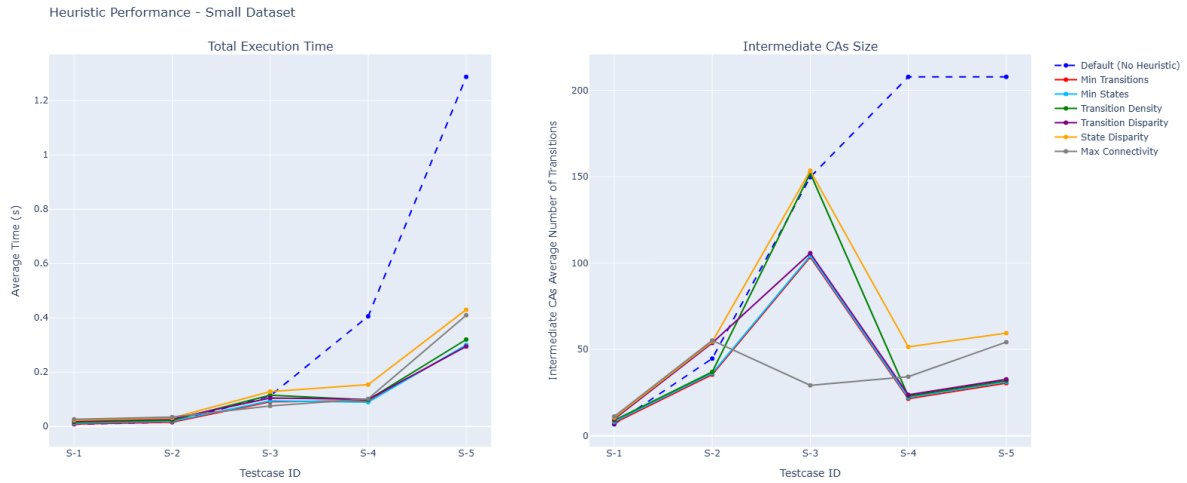


Figure 6: Comparison of the default approach and six heuristic-based approaches on the Small test case set, based on execution time and the average number of transitions in intermediate CAs. The time unit is seconds.

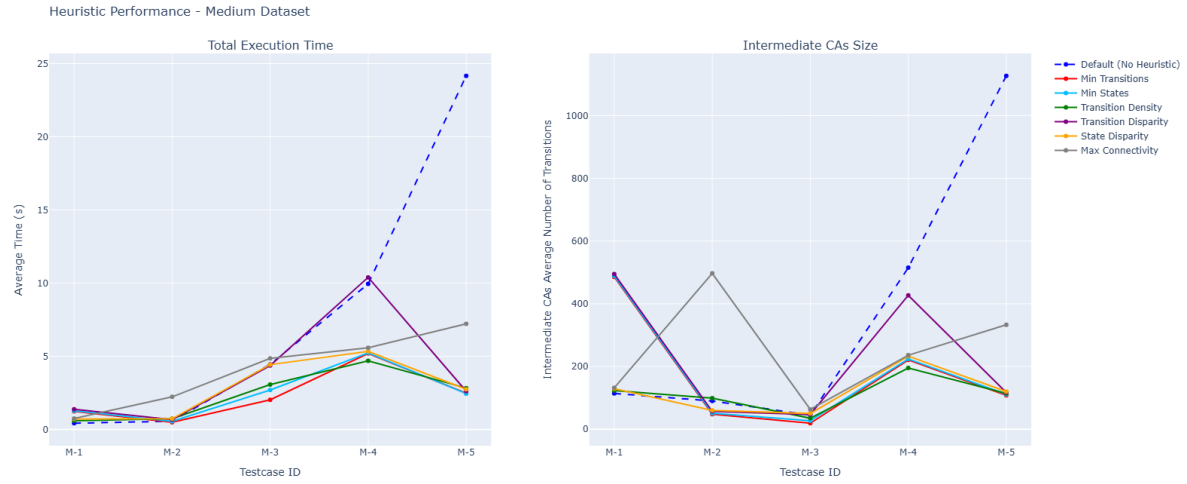


Figure 7: Comparison of the default approach and six heuristic-based approaches on the Medium test case set, based on execution time and the average number of transitions in intermediate CAs. The time unit is seconds.

The results indicate that the performance difference between the default approach and the heuristic-based approach becomes more noticeable as the number of states and transitions in the input automata increases. A comparison between the size of intermediately generated CAs in each joining sequence and the total execution time confirms a clear connection between the size of intermediately generated CAs and the operation's performance. Minimizing the size of intermediate CAs improves the performance of the join operation.

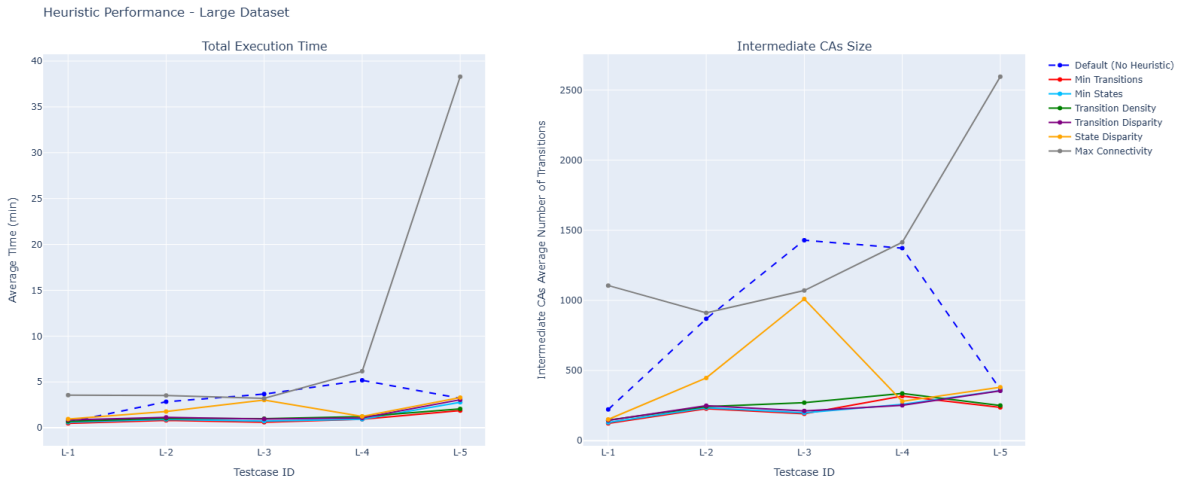


Figure 8: Comparison of the default approach and six heuristic-based approaches on the Large test case set, based on execution time and the average number of transitions in intermediate CAs. The time unit is minutes.

4.3 Discussion

The impact of proposed heuristics on performance is not the same for every scenario, and we were unable to find a single heuristic to be the most efficient for all scenarios. However, we can characterize the input automata by identifying a set of features, and select an appropriate heuristic based on the features of the remaining CAs to be joined. We will discuss some features in the suggested heuristics that may cause the generation of inefficient join orders, along with some features in the CAs that can help us identify whether a heuristic is efficient or not.

Min Transitions Heuristic. As shown by our results, this heuristic is generally effective, since the number of transitions can be considered a good indicator to measure the size of the CAs. However, it may not always work well. Suppose that we have three CAs and we want to pick the two smallest automata to join. The heuristic first identifies a CA named CA1 (not to be confused with CA1 in Section 3) as the one with the fewest transitions. It then needs to decide between two CAs, CA2 and CA3, to join with CA1. We observe that CA2 has a total of n transitions and m states, while CA3 has slightly fewer transitions but significantly more states. Therefore, the heuristic selects CA3 over CA2 and joins it with CA1. However, CA2 would have been a better choice, as it has significantly fewer states than CA3. This leads to the concepts of *density* and *sparsity* in automata when modeled as graphs, where *dense* CAs have a high number of transitions per state compared to *sparse* CAs.

Min States Heuristic. Similar to the previous heuristic, this one considers the number of states in an automaton as its size indicator and may encounter the same issues. Specifically, it may choose a CA with fewer states but a large number of transitions over one with slightly more states but significantly fewer transitions. This heuristic is generally accurate when the numbers of transitions and states in all CAs are close together, but it may not always generate the efficient join order. Considering additional heuristics, such as *max connectivity* or *transition density*, alongside this approach may improve the selection accuracy, especially when multiple CAs have similar state and transition counts.

Transition Density Heuristic. This heuristic first selects CAs with a low number of

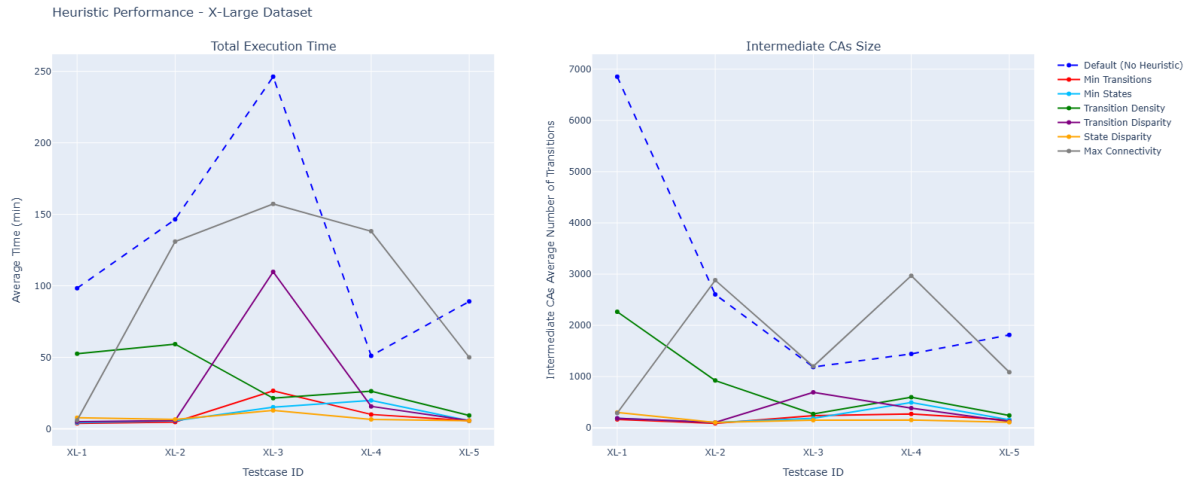


Figure 9: Comparison of the default approach and six heuristic-based approaches on the X-Large test case set, based on execution time and the average number of transitions in intermediate CAs. The time unit is minutes.

transitions per state. However, the density of a CA is not necessarily an indicator of its number of states and transitions, so this heuristic may fail to generate an efficient join sequence, particularly when dealing with CAs that vary in their state and transition counts. It would be more effective to combine this heuristic with others rather than using it as a standalone approach.

Transition Disparity Heuristic. This heuristic prioritizes joining CAs that are close to each other by their number of transitions to avoid the generation of large intermediate CAs. Since this heuristic only looks for a pair of CAs with the lowest disparity and does not analyze their own features, such as each CA's size, it might generate an inefficient sequence. It would be a better option to use this heuristic when all CAs are close to each other in size (number of transitions) or to combine it with other heuristics, such as the *minimum transitions* heuristic, to select a pair of CAs that are small and close to each other in size.

State Disparity Heuristic. Similar to the previous heuristic, this heuristic also does not consider the feature of each CA but only its relation to other CAs. Therefore, it would be better to combine it with other heuristics instead of using it as a standalone option to avoid generating inefficient join sequences.

Max Connectivity Heuristic. As previously mentioned, this heuristic was proposed to select CAs that are more connected to each other in the original Reo circuit to minimize the number of possible transitions in the resulting CA. Since this heuristic can greatly affect the size of the resulting CA, it would be a good idea to use it when the CAs are close to each other in size. However, when dealing with CAs diverse in size, we must not solely rely on it. As our test cases show, this heuristic generates highly inefficient join sequences for such cases compared to other heuristics.

Using *minimum transitions* or *minimum states* heuristic will select smaller CAs as the input for the *SingleJoin* algorithm. The size of the resulting CA of the *SingleJoin* algorithm not only depends on the size of the input CAs but also on the transition labels they share. Shared transition labels can cause the resulting CA to have a lower number of

transitions. Therefore, combining the two previously suggested heuristics with the *Max Connectivity* heuristic can be used as the backbone of an effective approach. We can also combine this approach with the other LLM-suggested heuristics to further enhance our accuracy in selecting an efficient joining sequence.

4.3.1 Combining Heuristics

To reduce the shortage of an individual heuristic, different heuristics can be combined together as suggested in the previous section. We consider three combined heuristics using the *Min Transitions*, *Min States*, and *Max Connectivity* as the backbone of our combinations. The first two are chosen because they serve as the primary indicators of the size of an automaton and have performed well in test cases. *Max Connectivity* is also included as it is shown to be effective in our experiments when the CAs are close to each other in their number of states and transitions. Joining two CAs with more shared transition labels generally results in a smaller CA compared to joining CAs with the same numbers of states and transitions but fewer shared labels [9]. Our generated heuristics are as follows:

Transitions and States Product Heuristic. This heuristic combines the *States Count* and *Transitions Count* heuristics by computing the product of a CA's state and transition counts, treating both as indicators of its size. It treats these factors as equal contributors to an automaton's size, aiming to minimize their individual shortage. Since all CAs in our running example have the same number of states, this heuristic produces the same joining sequence as the *Minimum Transitions* heuristic, as shown in Figure 5b.

Transitions and Connectivity Product Heuristic. This heuristic combines the *Min Transitions* and *Max Connectivity* heuristics. It first analyzes all possible pairs of CAs and computes the product of their transition counts. This value is then inverted and multiplied by the number of shared transition labels between the two CAs. The resulting value serves as the heuristic score for each pair. After computing scores for all pairs, the pair with the highest score is selected, joined, and the resulting CA is added to the list of remaining CAs. In our running example, CA1, CA2, and CA3 share a common transition label (X), while CA3 and CA4 share another label (Z). Among these pairs, (CA1, CA3), (CA2, CA3), and (CA3, CA4) have the lowest transition number product, each equal to 2. From these, CA1 and CA3 are selected for joining, resulting in a CA with one state and two transitions. This leaves three CAs, each with one state and two transitions. Next, CA1 $\bowtie$ CA3 and CA2 are selected for joining. The final join sequence produced by this heuristic is (((CA1 $\bowtie$ CA3) $\bowtie$ CA2) $\bowtie$ CA4), the same as the sequence shown in Figure 5b.

States and Connectivity Product Heuristic. This heuristic is similar to the previous one, but instead of transitions, it computes the product of the state counts for each pair of CAs, then inverts the result and multiplies it by the number of shared transition labels. In our running example, since all CAs (including the intermediate ones) have exactly one state, this heuristic produces the same joining sequence as the *Max Connectivity* heuristic, which matches the default sequence shown in Figure 5a.

Figures 10 and 11, show the computation time and average number of intermediate CAs transitions for the Medium and X-Large datasets, respectively, where the *Min Transitions*, *Min States*, and the *Max Connectivity* heuristics are compared with these new combined heuristics and the default approach in both of these criteria. To enhance the

visualization of our plot, a small jitter value is added to the plotted values to prevent the lines from overlapping. As seen in the results, the combined heuristics (particularly the *Transitions and Connectivity Product* and the *States and Connectivity Product* heuristics) have performed well and can be used as effective heuristics for joining constraint automata. By minimizing the individual shortage of single heuristics, these combined approaches can significantly improve the join operation's performance.

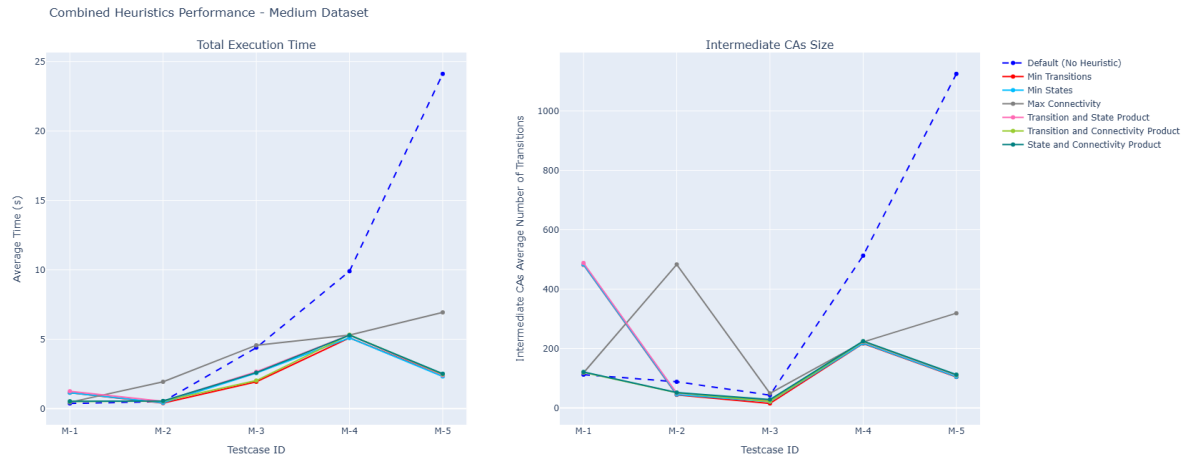


Figure 10: Comparison of the default approach, *Min Transitions*, *Min States*, *Max Connectivity*, and the three combined heuristics on the Medium test case set, based on execution time and the average number of transitions in intermediate CAs. The time unit is seconds.

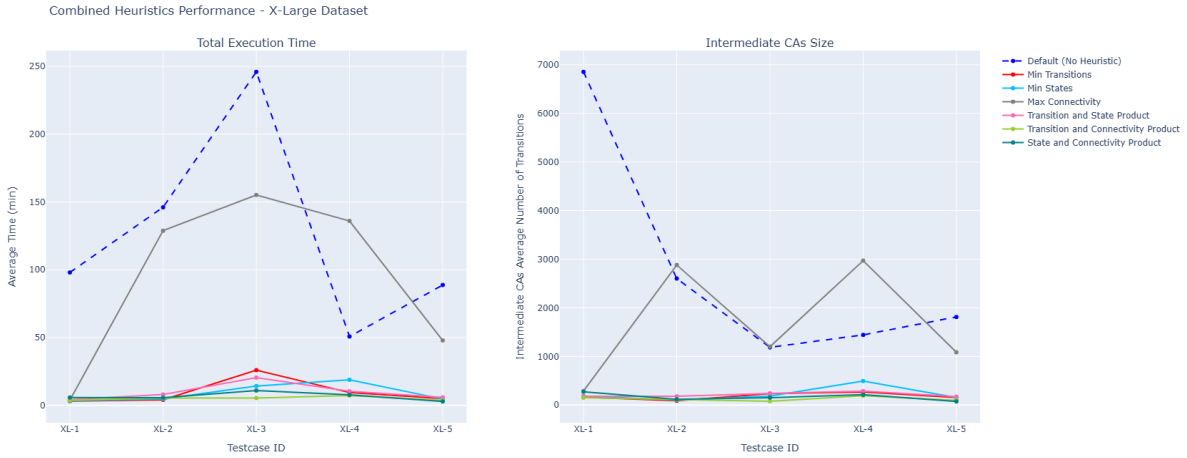


Figure 11: Comparison of the default approach, *Min Transitions*, *Min States*, *Max Connectivity*, and the three combined heuristics on the X-Large test case set, based on execution time and the average number of transitions in intermediate CAs. The time unit is minutes.

5 Conclusion and Future Work

In this paper, we implemented and evaluated multiple LLM-suggested heuristics for improving the efficiency of the join operation on multiple constraint automata, reducing the size of intermediately generated compounds.

To further improve this operation, we can employ more advanced methods for the heuristic selection while also leveraging the previously proposed optimization techniques for the join algorithm itself [6, 9]. For the former concern, we can either practice different possible combination of heuristics to more accurately consider different features of constraint automata or leverage more advanced machine learning-based approaches for dynamic heuristic selection based on the input automata features. We can combine these approaches with parallel programming and concurrency techniques to further enhance the operation's efficiency.

References

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

[2] F. Arbab, C. Baier, J. Rutten, and M. Sirjani. Modeling component connectors in reo by constraint automata. In *Proceedings of Second International Workshop on Foundations of Coordination Languages and Software Architectures (FLOCASA'03)*, volume 97, pages 25–46, July 2004.

[3] Christel Baier. Probabilistic models for reo connector circuits. *Journal of Universal Computer Science*, 11(10):1718–1748, 2005.

[4] A. Ghadiri. A tool for constraint automata join, bs project. Technical report, ECE Department University of Tehran, 2004.

[5] F. Ghassemi, S. Tasharofi, and M. Sirjani. Automated mapping of reo circuits to constraint automata. In *FSEN 2005*, volume 159 of *ENTCS*, pages 99–115. 2006.

[6] S. S. T. Q. Jongmans, T. Kappé, and F. Arbab. Composing constraint automata, state-by-state. In C. Braga and P. Ölveczky, editors, *Formal Aspects of Component Software. FACS 2015*, volume 9539 of *Lecture Notes in Computer Science*. Springer, Cham, 2016. doi: 10.1007/978-3-319-28934-2_12.

[7] N. Mehta, M. Sirjani, and F. Arbab. Effective modeling of software architectural assemblies using constraint automata. Technical Report SEN-R0309, CWI, Amsterdam, The Netherlands, 2003.

[8] N. Mehta, N. Medvidovic, M. Sirjani, and F. Arbab. Modeling behavior in compositions of software architectural primitives. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 371–374, Linz, Austria, September 2004. IEEE Computer Society.

[9] B. Pourvatan and N. Rouhy. An alternative algorithm for constraint automata product. In *FSEN 2007*, volume 4767 of *LNCS*, pages 412–422. 2007.

[10] Samira Tasharofi, Mohsen Vakilian, Reza Z. Moghaddam, and Marjan Sirjani. Modeling web service interactions using the coordination language reo. In *Proceedings of the International Workshop on Web Services and Formal Methods*, volume 4937 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2008.