Automata-based Optimization of Interaction Protocols for Scalable Multicore Platforms

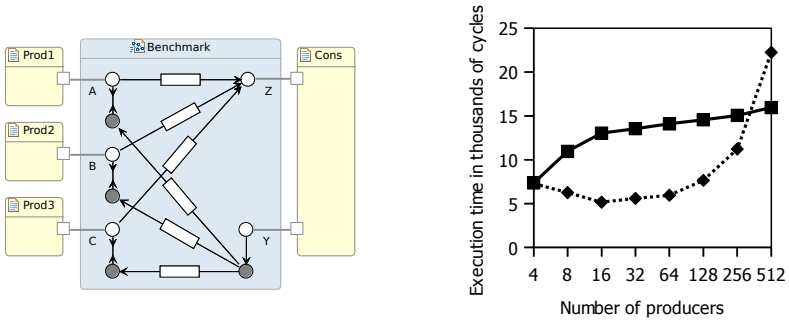
Sung-Shik T.Q. Jongmans, Sean Halle, and Farhad Arbab

Centrum Wiskunde & Informatica, Amsterdam, Netherlands
[jongmans,sean,farhad]@cwi.nl

Abstract. Multicore platforms offer the opportunity for utilizing massively parallel resources. However, programming them is challenging. We need good compilers that optimize commonly occurring synchronization/interaction patterns. To facilitate optimization, a programming language must convey what needs to be done in a form that leaves a considerably large decision space on *how* to do it for the compiler/run-time system.
Reo is a coordination-inspired model of concurrency that allows compositional construction of interaction protocols as declarative specifications. This form of protocol programming specifies only *what* needs to be done and leaves virtually all *how*-decisions involved in obtaining a concrete implementation for the compiler and the run-time system to make, thereby maximizing the potential opportunities for optimization. In contrast, the imperative form of protocol specification in conventional concurrent programming languages, generally, restrict implementation choices (and thereby hamper optimization) due to overspecification.
In this paper, we use the Constraint Automata semantics of Reo protocols as the formal basis for our optimizations. We optimize a generalization of the producer-consumer pattern, by applying CA transformations and prove the correctness of the transforms.

1 Introduction

Context. Coordination languages have emerged for the implementation of protocols among concurrent entities (e.g., threads on multicore hardware). One such language is Reo [1,2], a graphical language for compositional construction of *connectors* (i.e., custom synchronization protocols). Figure 1a shows an example. Briefly, a connector consists of one or more edges (henceforth referred to as *channels*), through which data items flow, and a number of nodes (henceforth referred to as *ports*), on which channel ends coincide. The connector in Figure 1a contains three different channel classes, including standard synchronous channels (normal arrows) and asynchronous channels with a buffer of capacity 1 (arrows decorated with a white rectangle, which represents a buffer). Through connector *composition* (the act of gluing connectors together on their shared ports), programmers can construct arbitrarily complex connectors. As Reo supports both synchronous and asynchronous channels, connector composition enables mixing synchronous and asynchronous communication within the same specification.



(a) Connector

(b) Per-interaction overhead for the Pthreads-based implementation (continuous line; squares) and the pre-optimized CA-based implementation (dotted line; diamonds)

Fig. 1: Producers–consumer benchmark

Especially when it comes to multicore programming, Reo has a number of advantages over conventional programming languages with a fixed set of low-level synchronization constructs (locks, mutexes, etc.). Programmers using such a conventional language have to translate the synchronization needs of their protocols into the synchronization constructs of that language. Because this translation occurs in the mind of the programmer, invariably some context information either gets irretrievably lost or becomes implicit and difficult to extract in the resulting code. In contrast, Reo allows programmers to compose their own synchronization constructs (i.e., connectors) at a high abstraction level to perfectly fit the protocols of their application. Not only does this reduce the conceptual gap for programmers, which makes it easier to implement and reason about protocols, but by preserving all relevant context information, such user-defined synchronization constructs also offer considerable novel opportunities for compilers to do optimizations on multicore hardware. This paper shows one such occasion.

Additionally, Reo has several software engineering advantages as a domain-specific language for protocols [3]. For instance, Reo forces developers to separate their computation code from their protocol code. Such a separation facilitates verbatim reuse, independent modification, and compositional construction of protocol implementations (i.e., connectors) in a straightforward way. Moreover, Reo has a strong mathematical foundation [4], which enables formal connector analyses (e.g., deadlock detection, model checking [5]).

To use connectors in real programs, developers need tools that automatically generate executable code for connectors. In previous work [6], we therefore developed a Reo-to-C compiler, based on Reo's formal semantics of *constraint automata* (CA) [7]. In its simplest form, this tool works roughly as follows. First,

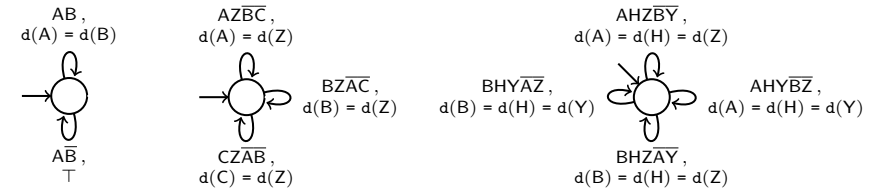
it extracts from an input XML representation of a connector a list of its primitive constituents.[1] Second, it consults a database to find for every constituent in the list a "small" CA that formally describes the behavior of that particular constituent. Third, it computes the product of the CA in the constructed collection to obtain one "big" CA describing the behavior of the whole connector. Fourth, it feeds a data structure representing that big CA to a template. Essentially, this template is an incomplete C file with "holes" that need be "filled". The generated code simulates the big CA by repeatedly computing and firing eligible transitions in an event-driven fashion. It runs on top of Proto-Runtime [8,9], an execution environment for C code on multicore hardware. A key feature of Proto-Runtime is that it provides more direct access to processor cores and control over scheduling than threading libraries based on OS threads, such as Pthreads [10].

Problem. Figure 1a shows a connector for a protocol among $k = 3$ producers and one consumer in a producers–consumer benchmark. Every producer loops through the following steps: (i) it produces, (ii) it blocks until the consumer has signaled ready for processing the next batch of productions, and (iii) it sends its production. Meanwhile, the consumer runs the following loop: (i) it signals ready, and (ii) it receives exactly one production from every producer in arbitrary order. We compared the CA-based implementation generated by our tool with a hand-crafted implementation written by a competent C programmer using Pthreads, investigating the time required for communicating a production from a producer to the consumer as a function of the number of producers.

Figure 1b shows our results. On the positive side, for $k \leq 256$, the CA-based implementation outperforms the hand-crafted implementation. For $k = 512$, however, the Pthreads-based implementation outperforms the generated implementation. Moreover, the dotted curve looks disturbing, because it grows more-than-linearly in k: indeed, the CA-based implementation scales poorly. (We skip many details of this benchmark, including those of the Pthreads-based implementation, and the meaning/implications of these experimental results. The reason is that this paper is *not* about this benchmark, and its details do not matter. We use this benchmark only as a concrete case to better explain problems of our compilation approach and as a source of inspiration for solutions.)

Contribution. In this paper, we report on work at improving the scalability of code generated by our Reo-to-C compiler. First, we identify a cause of poor scalability: briefly, computing eligibility of k transitions in producers–consumer-style protocols (and those generalizations thereof that allow any synchronization involving one party from every one of ℓ groups) takes $\mathcal{O}(k)$ time instead of $\mathcal{O}(1)$, of which the Pthreads-based implementation shows that it is possible. Second, to familiarize the reader with certain essential concepts, we explain a manual solution (in terms of Reo's CA semantics) that achieves $\mathcal{O}(1)$. Third, we propose an automated, general solution, built upon the same concepts as the manual

[1] Programmers can use the ECT plugins for Eclipse (`http://reo.project.cwi.nl`) to draw connectors such as the one in Figure 1a, internally represented as XML.



(a) LossySync (b) Merger$_3$ (c) Hourglass

Fig. 2: Example CA, called LossySync, Merger$_3$ and Hourglass

solution. We formalize this automated solution and prove it correct. Although inspired by our work on Reo and formulated in terms of CA, we make more general contributions beyond Reo and CA, better explained in our conclusion.

We organized the rest of this paper as follows. In Section 2, we explain CA. In Section 3, we analyze how the Pthreads-based implementation avoids scalability issues and how we can export that to our setting. In Sections 4–6, we automate the solution proposed in Section 3. Section 7 concludes this paper. Definitions and detailed proofs appear in the appendix of a technical report [11].

Although inspired by Reo, we can express our main results in a purely automata-theoretic setting. We therefore skip a primer on Reo [1,2].

2 Constraint Automata

Constraint automata are a general formalism for describing systems behavior and have been used to model not only connectors but also, for instance, actors [12]. Figure 2 shows examples.[2,3] In the context of this paper, a CA specifies *when* during execution of a connector *which* data items flow *where*. Structurally, every CA consists of finite sets of states, transitions between states, and ports. States represent the internal configurations of a connector, while transitions describe its atomic execution steps. Every transition has a label that consists of two elements: a *synchronization constraint* (SC) and a *data constraint* (DC). An SC is a propositional formula that specifies which ports synchronize in a firing transition

[2] The LossySync CA models a connector with one input port A and one output port B. It repeatedly chooses between two atomic execution steps (constrained by availability of pending I/O operations): synchronous flow of data from A to B or flow of data only on A (after which the data is lost, before reaching B). The Merger$_3$ CA models a connector with three input ports A, B, and C and one output port Z. It repeatedly chooses between three atomic execution steps: synchronous flow of data from A to Z, from B to Z, or from C to Z. Finally, the Hourglass CA models a connector with two input ports A and B, one internal port H, and two output ports Y and Z. It repeatedly chooses between four atomic execution steps: synchronous flow of data from A via H to Y, from A via H to Z, from B via H to Y, or from B via H to Z.

[3] We show only single state CA for simplicity. Generally, a CA can have any finite number of states, and the results in this paper are applicable also to such CA.

$$
\begin{array}{l}
p \ ::= \text{ any element from } \mathbb{P}\textsc{ort} \\
\Psi \ ::= \text{ any set of } \textsc{sc}\text{s} \\
a \ ::= 0 \mid 1 \mid p \\
\psi \ ::= a \mid \overline{\psi} \mid \psi + \psi \mid \psi \cdot \psi \mid \bigoplus(\Psi)
\end{array}
$$

(a) Synchronization constraints

$$
\begin{array}{l}
p \ ::= \text{ any element from } \mathbb{P}\textsc{ort} \\
P \ ::= \text{ any subset of } \mathbb{P}\textsc{ort} \\
b \ ::= \bot \mid \top \mid \mathtt{Eq}(P) \mid \mathtt{d}(p) = \mathtt{d}(p) \\
\phi \ ::= b \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi
\end{array}
$$

(b) Data constraints

Fig. 3: Syntax

(i.e., where data items flow); a DC is a propositional formula that (under)specifies which particular data items flow where. For instance, in Figure 2a, the DC $\mathtt{d}(\mathsf{A}) = \mathtt{d}(\mathsf{B})$ means that the data item on A equals the data item on B; the DC $\top$ means that it does not matter which data items flow. Let $\mathbb{P}\textsc{ort}$ denote the global set of all ports. Formally, an SC is a word ψ generated by the grammar in Figure 3a, while a DC is a word ϕ generated by the grammar in Figure 3b.

Figure 3a generalizes the original definition of SCs as sets of ports interpreted as conjunctions [7] (shortly, we elaborate on the exact correspondence). Operator $\bigoplus$ is a uniqueness quantifier: $\bigoplus(\Psi)$ holds if exactly one SC in Ψ holds. Also, we remark that predicate $\mathtt{Eq}(P)$ is novel. It holds if equal data items are distributed over all ports in P. In many practical cases—but not all—we can replace a DC of the shape $\mathtt{d}(p_1) = \mathtt{d}(p_2)$ with $\mathtt{Eq}(P)$ if $\{p_1, p_2\} \subseteq P$. In the development of our optimization technique, $\mathtt{Eq}(P)$ plays an important role (see also Section 7).

Let $\mathbb{D}\textsc{ata}$ denote the set of all data items. Formally, we interpret SCs and DCs over *distributions* of data over ports, $\delta : \mathbb{P}\textsc{ort} \rightharpoonup \mathbb{D}\textsc{ata}$, using relations $\models^{\text{sc}}$ and $\models^{\text{dc}}$ and the corresponding equivalence relations $\equiv_{\text{sc}}$ and $\equiv_{\text{dc}}$. Their definition for negation, disjunction, and conjunction is standard; for atoms, we have:

$$
\delta \models^{\text{sc}} p \ \textbf{iff} \ p \in \text{Dom}(\delta) \qquad \begin{array}{ll} \delta \models^{\text{dc}} \mathtt{Eq}(P) & \textbf{iff} \ |\text{Img}(\delta|_P)| = 1 \\ \delta \models^{\text{dc}} \mathtt{d}(p_1) = \mathtt{d}(p_2) & \textbf{iff} \ \delta(p_1) = \delta(p_2) \end{array}
$$

Let $\sum(\{\psi_1, \ldots, \psi_k\})$ and $\prod(\{\psi_1, \ldots, \psi_k\})$ abbreviate $\psi_1 + \cdots + \psi_k$ and $\psi_1 \cdots \psi_k$, let $\mathbb{SC}$ denote the sets of all SCs, and let $\mathsf{SC}(P)$ and $\mathsf{DC}(P)$ denote the sets of all SCs and all DCs over ports in P.

A constraint automaton is a tuple $(Q, P, \longrightarrow, \imath)$ with Q a set of states, $P \subseteq \mathbb{P}\textsc{ort}$ a set of ports, $\longrightarrow \subseteq Q \times \mathsf{SC}(P) \times \mathsf{DC}(P) \times Q$ a transition relation labeled with $[\text{SC}, \text{DC}]$-pairs of the form (ψ, ϕ), and $\imath \in Q$ an initial state.

A distribution δ represents a single atomic execution step of a connector in which data item $\delta(p)$ flows on port p (for all ports in the domain of δ). A CA α accepts *streams* (i.e., infinite sequences) of such distributions. Every such a stream represents one possible infinite execution of the connector modeled by α. Intuitively, to see if α accepts a stream σ, starting from the initial state, take the first element $\sigma(0)$ from the stream, check if α has a (ψ, ϕ)-labeled transition from the current state such that $\sigma(0) \models^{\text{sc}} \psi$ and $\sigma(0) \models^{\text{dc}} \phi$, and if so, make this transition, remove $\sigma(0)$ from the stream, and repeat.

Our CA definition generalizes the original definition of CA [7], because Figure 3a generalizes the original definition of SCs. However, CA as originally defined

(a) Current (b) Manual improvement (c) Automated improvement

Fig. 4: Code generation diagrams

still play a role in the development of our optimization technique: all input CA that this technique operates on are original. Therefore, we make more precise what "originality" means. First, let a *P-complete product* be a product of either a positive or a negative literal for every port in P. Intuitively, a P-complete product specifies not only which ports participate in a transition, but it also makes explicit which ports idle in that transition. Let $\mathtt{cp}(P\,,\,P_+)$ denote a P-complete product with positive literals $P_+ \subseteq P$. Then, we call an SC ψ original if a set P_+ exists such that $\mathtt{cp}(P\,,\,P_+) \equiv \psi$ (originally, set P_+ would be the SC); we call a CA original if it has only original SCs. All CA in Figure 2 are original.

We adopt *bisimilarity* on CA as behavioral congruence, derived from the definition for original CA of Baier et al. [7]. Roughly, if α and β are bisimilar, denoted as $\alpha \sim \beta$, α can simulate every transition of β in every state and vice versa (see Definition 32 in [11, Appendix A]).

3 Enhancing Scalability: Problem and Solution

We study the scalability of code generated by our compiler using Figure 4. We start with Figure 4a, which summarizes the code generation process of our current tool: given an original CA α (computed for the connector to generate code for), it generates a piece of code `alpha` by applying transformation g_1.

Essentially, `alpha` consists of an event-driven *handler*, which simulates α. This handler runs concurrently with the code of its environment (i.e., the code of the entitites under coordination), whose events (i.e., I/O operations performed on ports) it listens and responds to, as follows. Whenever the environment performs an I/O operation on a port p, it assigns a representation of that operation to an `event` variable in a data structure for p (also generated by transformation g_1 and part of `alpha`). This causes the handler to start a new round of simulating α. Based on the state of α that the handler at that point should behave as, the handler knows which transitions of α *may* fire. Which of those transitions actually *can* fire, however, depends also on the pending events that previously occurred (i.e., the pending I/O operations on ports). To investigate this, the handler checks *for every transition* that may fire if the pending events (including the new one) can constitute a distribution δ that satisfies the transition's label. If so, the handler fires the transition: it distributes data over ports according to δ, and the events involved dissolve. Otherwise, if no transition can fire, all events remain for the next round, and the handler goes dormant.

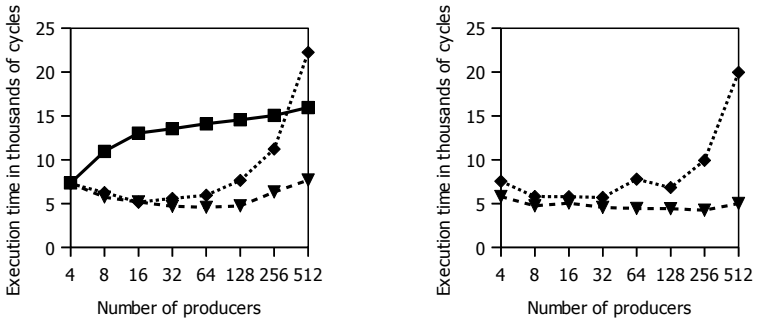
Now, recall our producers–consumer benchmark in Section 1. Figure 2b shows the CA for the connector in Figure 1a.[4] Generally, for an arbitrary number of producers k, the corresponding CA α_k has k transitions. Consequently, in the worst case, the handler in the generated `alpha_k` code performs k checks in every event handling round, which takes $\mathcal{O}(k)$ time. Figure 1b shows this as a more-than-linear increase in execution time for the dotted curve.[5] The Pthreads-based implementation, in contrast, uses a queue for lining up available productions. To receive a production, the consumer simply dequeues, which takes only $\mathcal{O}(1)$ time (ignoring, for simplicity, the overhead of synchronizing queue accesses). Figure 1b shows this as a linear increase in execution time for the continuous curve.

Intuitively, by checking all transitions to make the consumer receive, the generated CA-based implementation performs an exhaustive search for a particular producer that sent a production. In contrast, by using a queue, the Pthreads-based implementation avoids such a search: the queue embodies that in this protocol, it does not matter which *particular* producer sent a production as long as *some* producer has done so (in which case the queue is nonempty). The producers are really *indistinguishable* from the perspective of the consumer. Thus, to improve the scalability of code generated by our tool, we want to export the idea of "using queues to leverage indistinguishability" to our setting.

Figure 4b shows a first attempt at achieving this goal: we introduce a manual transformation h that takes `alpha` as input and hacks together a new piece of code `beta`, which should (i) behave as `alpha`, (ii) demonstrate good scalability, and (iii) use queues. For instance, in our producers–consumer example ($k = 3$), h works roughly as follows. First, h replaces the `event` variable in the data structure for every port $p \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{Z}\}$ with an `eventQueue` variable that points to a queue of pending events. In this new setup, to perform an I/O operation, the environment enqueues an `eventQueue`, while handler code tests `eventQueue`s for nonemptiness to check SCs, peeks `eventQueue`s to check DCs, and dequeues `eventQueue`s to fire transitions. Subsequently, h adds initialization code to `alpha` to ensure that the `eventQueue` variables of ports A, B, and C all point to the same shared queue, while the `eventQueue` variable for port Z points to a different queue. Here, h effectively exploits the indistinguishability property of producers by making *the ports* that those producers use indistinguishable in our setting. Finally, h updates the handler code such that it processes the shared queue only

[4] To be precise, the CA in Figure 2b describes the behavior of one of the *synchronous regions* of the connector in Figure 1a (i.e., a particular subconnector of the whole). This point is immaterial to our present discussion, however, and ignoring it simplifies our presentation without loss of generality or applicability.

[5] The growth is more-than-linear instead of just linear because of the barrier in the protocol. When producer P is ready to send its $(i+1)$-th production, the consumer may not yet have received the i-th production from all other producers. Then, P must wait until the consumer signals ready (i.e., the barrier). In the worst case, however, the consumer has received an i-th production only from P such that P has to wait $(k-1) \cdot \mathcal{O}(k)$ time. Afterward, it takes another $\mathcal{O}(k)$ time for P to send its $(i+1)$-th production. Consequently, sending the $(i+1)$-th production takes $k \cdot \mathcal{O}(k)$ time, and the complexity of sending a production lies between $\mathcal{O}(k)$ and $\mathcal{O}(k^2)$.



(a) With barrier (b) Without barrier

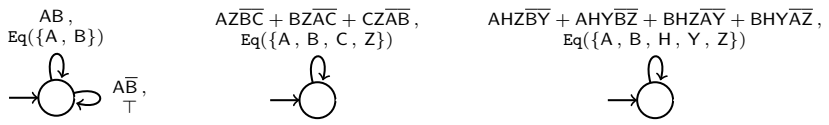
Fig. 5: Per-interaction overhead for the Pthreads-based implementation (continuous line; squares), the pre-optimized CA-based implementation (dotted line; diamonds), and the optimized, h-transformed CA-based implementation (dashed line; triangles) of the producers–consumer scenario in Figure 1

once per event handling round instead of thrice (i.e., once for every transition). From an automata-theoretic perspective, h replaces the implementation of the three "physical" transitions with an implementation of one merged "virtual" transition. When the handler fires this virtual transition at run-time, it actually fires one of the three physical transitions.

Property (iii) holds of the piece of code `beta` resulting from applying h to `alpha` as just described. Figure 5 shows that also property (ii) holds. The dashed curve in Figure 5a shows execution times of h-transformed code of the CA-based implementation in the producers–consumer benchmark. The h-transformed code scales much better than the original code. Additionally, Figure 5b shows execution times of the producers–consumer benchmark without a barrier (i.e., producers send productions whenever they want). In this variant, h achieves even better results: it transforms a poorly scalable program into one that scales perfectly.[6]

Establishing property (i), however, is problematic. Although we can informally argue that it holds, proving this—formally showing the equivalence of two concurrent C programs—seems prohibitively complex. That aside, the manual nature of h makes its usage generally impractical, and it seems extremely difficult to automate it: an automated version of h would have to analyze C code to recover relevant context information about the protocol, which is not only hard but often theoretically impossible. Similarly, it seems infeasible to write an

[6] Of course, in many cases and for many applications, a purely asynchronous producers–consumer protocol without a barrier, as in Figure 5b, suffices. The reason that we initially focused on a producers–consumer protocol with a barrier, which is also useful yet in other applications, is that its mix of synchrony and asynchrony makes it a harder, and arguably more interesting, protocol to achieve good scalability for. Comparing the results in Figures 5a and 5b also shows this.



(a) LossySync (b) Merger$_3$ (c) Hourglass

Fig. 6: Application of transformations f_1 to the CA in Figure 2

optimizing compiler able to transform, for instance, less scalable Pthreads-based implementations of the producers-consumer scenario (without queues) into the Pthreads-based implementation (with queues) used in our benchmark. The inability of compilers for lower-level languages to do such optimizations seems a significant disadvantage of using such languages for multicore programming.

We therefore pursue an alternative approach, outlined in Figure 4c: we introduce a transformation f that takes CA α as input—instead of the low-level C code generated for it—and transforms it into an equivalent automaton β, a variant of α with *merged transitions* (cf. transformation h, which implicitly replaced the implementation of several physical transitions with one virtual transition). Crucially, α still explicitly contains all relevant context information about the protocol, exactly what makes f eligible to automation. In particular, to merge transitions effectively, f carefully inspects transition labels and takes port indistinguishability into account. The resulting merged transitions have an "obvious" and mechanically obtainable implementation using queues. A subsequent transformation g_2, from β to `beta`, performs this final straightforward step.

We divide transformation $\alpha \xrightarrow{f} \beta$ into a number of constituent transformations $\alpha \xrightarrow{f_1} \beta' \xrightarrow{f_2} (\beta' , \Gamma) \xrightarrow{f_3} \beta$, discussed in detail in the following sections.

4 Transformation f_1: Preprocessing

Transformation f_1 aims at merging transitions $t_1 , \ldots , t_k$ into one transition $(q , \psi , \texttt{Eq}(P) , q')$, where $\psi = \sum(\{\psi_1 , \ldots , \psi_k\})$. It consists of two steps.

In the first step, transformation f_1 replaces DCs on transitions of $\alpha = (Q , P , \longrightarrow , \imath)$ with $\texttt{Eq}(P)$, as follows. Because α is an original CA (our current code generator can handle only original CA), every SC in α is an original SC: for every transition label (ψ , ϕ), a set of ports P_+ exists such that $\texttt{cp}(P , P_+) \equiv \psi$. Now, for every product in *disjunctive normal form* (DNF) of ϕ, transformation f_1 constructs a graph with vertices P_+ and an edge (p_1 , p_2) for every $\texttt{d}(p_1) = \texttt{d}(p_2)$ literal. Because $\texttt{cp}(P , P_+) \equiv \psi$, if the resulting graph is connected, the product of the $\texttt{d}(p_1) = \texttt{d}(p_2)$ literals is equivalent to $\texttt{Eq}(P)$. Thus, f_1 replaces every transition label (ψ , ϕ) in α with an equivalent label (ψ , ϕ'), where ϕ' denotes the modified DNF of ϕ, with $\texttt{Eq}(P)$ for every product of $\texttt{d}(p_1) = \texttt{d}(p_2)$ literals if those literals induce a connected graph. Let α' denote the resulting CA. We can prove that $\alpha' \sim \alpha$ holds (see Lemma 16 in [11, Appendix A]).

In the second step, transformation f_1 merges, for every pair of states (q, q'), all transitions from q to q' labeled by DC ϕ into one new transition. (The individual transitions differ only in their SC.) Every resulting transition has as its SC the sum of the SCs of the individual transitions. Figure 6 shows examples. We denote the resulting CA by $f_1(\alpha)$. The following proposition holds, because choices between individual transitions in α are encoded in $f_1(\alpha)$ by sum-SCs of merged transitions. Consequently, α and $f_1(\alpha)$ can simulate each other's steps.

Proposition 1. $f_1(\alpha) \sim \alpha'$

5 Transformation f_2: Constructing Hypergraphs

Every merged transition resulting from the previous preprocessing transformations can *perhaps* be implemented using queues along the same lines as transformation h (see Section 3). In the first place, this depends on the extent to which ports in a merged transition are indistinguishable: no indistinguishable ports means no queues. Second, the SC of a merged transition should make port indistinguishability (i.e., queues), if present, apparent and *mechanically detectable*. The SCs of transitions in $f_1(\alpha)$ fail to do so. For instance, we (hence a computer) cannot directly derive from the syntax of SC $\mathsf{AZ\overline{BC}+BZ\overline{AC}+CZ\overline{AB}}$ in Figure 6b that its transition has a scalable implementation with queues. In contrast, the equivalent SC $\bigoplus(\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}) \cdot \mathsf{Z}$ makes this much more apparent. From this SC, we can "obviously" (and mechanically by transformation g_2 in Figure 4c) conclude that ports A, B, and C may share the same queue, from which exactly one element is dequeued per firing, because they are indistinguishable indeed: intuitively, if $\delta \overset{\text{sc}}{\models} \bigoplus(\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}) \cdot \mathsf{Z}$, we cannot know which one of A, B, or C holds, unless we inspect δ. Thus, beside automatically detecting indistinguishable ports in a transition, to actually reveal them as queues, we additionally need an algorithm for syntactically manipulating that transition's SC. We formulate both these aspects in terms of a per-transition *hypergraph* [13]. Working with hypergraph representations simplifies our reasoning about, and manipulation of, SCs modulo associativity and commutativity. We compute hypergraphs as follows.

Let $\alpha = (Q, P, \longrightarrow, \imath)$ be an original CA as before, and let (q, ψ, ϕ, q') be a (merged) transition in $f_1(\alpha)$. Because α is an original CA and by the construction of $f_1(\alpha)$, we know that ψ is a sum of P-complete products of ports (e.g., Figure 6). Because every single port p is equivalent to $\bigoplus(\{p\})$, transformation f_2 can represent ψ as a set $\mathcal{E}$ of sets E of sets V: $\mathcal{E}$ represents the outer sum, every E represents a P-complete product (E includes/excludes every positive/negative port), and every V represents an inner exclusive sum. For instance, $\{\{\{\mathsf{A}\}, \{\mathsf{Z}\}\}, \{\{\mathsf{B}\}, \{\mathsf{Z}\}\}, \{\{\mathsf{C}\}, \{\mathsf{Z}\}\}\}$ represents the SC of the transition in Figure 6b. Transformation f_2 considers $\mathcal{E}$ as the set of hyperedges of a hypergraph over the set of vertices $\wp(\mathsf{Port}(\psi))$, where $\mathsf{Port}(\psi)$ denotes the ports occurring in ψ (i.e., every vertex is a set of ports). Formally, f_2 computes a function graph. Let $\mathbb{G}$RAPH denote the set of all hypergraphs with sets of ports as vertices.

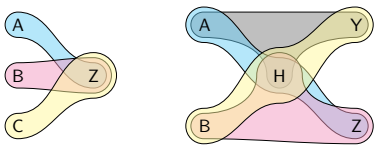
Definition 1. $\mathsf{graph} : \mathbb{SC} \rightharpoonup \mathbb{GRAPH}$ *denotes the partial function from* SCs *to hypergraphs defined as:*[7]

$$\mathsf{graph}(\psi) = (\wp(\mathsf{Port}(\psi))\,,\ \left\{ E \;\middle|\; \begin{array}{l} E = \{V \mid V = \{p\} \ \textbf{and}\ p \in P_+\} \\ \textbf{and}\ \ P_+ \subseteq \mathsf{Port}(\psi)\ \ \textbf{and}\ \ P_+ \in \mathcal{P} \end{array} \right\})$$

$$\textbf{if}\ \ \left[\psi = \sum(\left\{ \psi' \;\middle|\; \begin{array}{l} \psi' \equiv_{\mathrm{sc}} \mathsf{cp}(\mathsf{Port}(\psi)\,,\, P_+)\ \ \textbf{and} \\ P_+ \subseteq \mathsf{Port}(\psi)\ \ \textbf{and}\ \ P_+ \in \mathcal{P} \end{array} \right\})\ \ \textbf{for some}\ \ \mathcal{P}\right]$$

(The side condition states just that ψ is a sum of P-complete products of ports.)

Figure 7 shows example hypergraphs (without unconnected vertices).



(a) Merger_3 (b) Hourglass

Fig. 7: Hypergraphs for the transitions of the CA in Figure 6

We define the *meaning* of a hypergraph as a sum of products of exclusive sums, where every product corresponds to a hyperedge. Such a product consists of exclusive sums of positive ports (one for each vertex in the hyperedge), and it consists of negative ports (one for every port outside the vertices in the hyperedge). We can show that graph is an isomorphism (i.e., $\mathsf{graph}(\psi)$ is a sound and complete representation of ψ).

Definition 2. $[\![\cdot]\!] : (\wp(\mathbb{VER}) \times \wp(\mathbb{PORT})) \cup \mathbb{GRAPH} \to \mathbb{SC}$ *denotes the function from* $\big[$*hyperedge, set of ports*$\big]$*-pairs and hypergraphs to* SCs *defined as:*

$$\begin{aligned} [\![E]\!]_P \quad &= \textstyle\prod(\{\psi \mid \psi = \bigoplus(V)\ \ \textbf{and}\ \ V \in E\} \cup \\ &\qquad \{\psi \mid \psi = \overline{p}\ \ \textbf{and}\ \ p \in P \setminus (\bigcup E)\}) \\ [\![(\mathcal{V}\,,\,\mathcal{E})]\!] &= \textstyle\sum(\{\psi \mid \psi = [\![E]\!]_{\bigcup \mathcal{V}}\ \ \textbf{and}\ \ E \in \mathcal{E}\}) \end{aligned}$$

Theorem 1. *(Theorem 3 in [11, Appendix A])* $\psi \equiv [\![\mathsf{graph}(\psi)]\!]$

In summary, transformation f_2 computes graph for every merged transition in $f_1(\alpha)$ and stores each of those graphs in a set Γ (indexed by transitions).

Hypergraphs as introduced are generic representations of synchronization patterns, isomorphic to but independent of SCs in CA. This reinforces that our optimization approach, transformation f, is not tied CA but a generally applicable technique when relevant context information is available.

6 Transformation f_3: Manipulating SCs

Transformation f_3 aims at making all indistinguishable ports (hence queues) in SCs on (merged) transitions in $f_1(\alpha)$ apparent by analyzing and manipulating the hypergraphs in Γ, computed by transformation f_2. It consists of two steps.

In the first step, transformation f_3 computes the indistinguishable ports under every transition $t = (q\,,\,\psi\,,\,\phi\,,\,q')$ in $f_1(\alpha)$. We call the ports in a set I indistinguishable under t if for every distribution δ such that $\delta \overset{\mathrm{sc}}{\models} \psi$ and $|I \cap \mathrm{Dom}($

[7] Let $\wp(X)$ denote the power set of X.

$\delta)| = 1$, we cannot deduce from $\delta|_{P \setminus I}$ which particular port in I is satisfied by δ. An example appeared in the first paragraph of Section 5. In an implementation with a queue shared among the ports in I, this means that whenever t fires, we know that exactly one port in I participated in the transition but not which one, even if we know all other participating ports (i.e., those outside I).

By analyzing hypergraph $\gamma_t \in \Gamma$ for the SC ψ of t, transformation f_3 computes maximal sets of indistinguishable ports under t (larger sets of indistinguishable ports means larger queues means better scalability), as follows. Recall from Section 5 that γ_t represents a sum (hyperedge relation) of P-complete products (hyperedges) of singleton exclusive sums (vertices). To understand how port indistinguishability displays in γ_t, suppose that ports $p_1 , p_2 \in P$ are indistinguishable, and let δ be a distribution such that $\delta \overset{\text{sc}}{\models} [\![\gamma_t]\!]$. Because γ_t's hyperedge relation $\mathcal{E}$ represents a sum of P-complete products, exactly one hyperedge $E \in \mathcal{E}$ exists such that δ satisfies $[\![E]\!]_P$. Then, because $|\{p_1 , p_2\} \cap \text{Dom}(\delta)| = 1$, a vertex $V \in E$ exists such that $p_1 \in V$ or $p_2 \in V$.[8] In fact, because every hyperedge consists of singleton vertices, either $\{p_1\} \in E$ or $\{p_2\} \in E$. Now, by inspecting $\delta|_{P \setminus \{p_1, p_2\}}$, we can infer the other vertices in E, beside either $\{p_1\}$ or $\{p_2\}$. Let E' denote this set of vertices, and observe that either $E = E_1 = E' \uplus \{\{p_1\}\}$ or $E = E_2 = E' \uplus \{\{p_2\}\}$. Because both options are possible, $\mathcal{E}$ necessarily includes both E_1 and E_2, and importantly, E_1 and E_2 are equal up to p_1 and p_2.

Generalizing this example from $\{p_1 , p_2\}$ to arbitrarily sized sets I, informally, the ports in I are indistinguishable if every port in I is involved in the same hyperedges as every other port in I *up to occurrences of ports in* I. The following definitions make this generalization formally precise. First, we introduce a function Edge that determines for a port p which hyperedges in $\mathcal{E}$ include p. (In fact, $\mathsf{Edge}(p , \mathcal{E})$ contains all such hypergedges up to occurrences of vertices with p.) Then, we define a function $\bigstar$ that computes maximal sets of ports with the same set $\mathsf{Edge}(p , \mathcal{E})$. Importantly, $\bigstar$ yields a partition of the set of ports in vertices connected by $\mathcal{E}$, denoted by $\mathsf{Port}(\mathcal{E})$. Henceforth, we therefore call every maximal set of indistinguishable ports computed by $\bigstar$ a *part*.

Definition 3. $\mathsf{Edge} : \mathbb{P}\text{ORT} \times \wp^2(\mathbb{V}\text{ER}) \to \wp^2(\mathbb{V}\text{ER})$ *denotes the function from* $\big[$*port, set of hyperedges*$\big]$*-pairs to sets of hyperedges defined as:*

$$\mathsf{Edge}(p , \mathcal{E}) = \{\mathcal{W} \mid \mathcal{W} = E \setminus \{V\} \textbf{ and } p \in V \in E \in \mathcal{E}\}$$

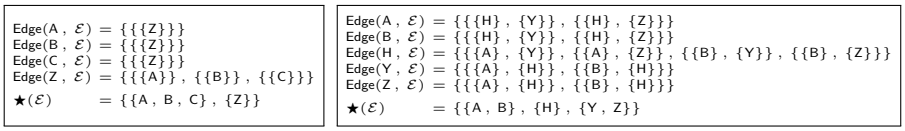
Definition 4. $\bigstar : \wp^2(\mathbb{V}\text{ER}) \to \wp^2(\mathbb{P}\text{ORT})$ *denotes the function from sets of hyperedges to sets of sets of ports defined as:*

$$\bigstar(\mathcal{E}) = \{P \mid P \in \wp^+(\mathsf{Port}(\mathcal{E})) \textbf{ and } \big[[p \in P \textbf{ iff } \mathcal{T} = \mathsf{Edge}(p , \mathcal{E})\big] \textbf{ for all } p\big]\}$$

Lemma 1. *(Lemma 12 in [11, Appendix A])*

1. $\bigcup \bigstar(\mathcal{E}) = \mathsf{Port}(\mathcal{E})$
2. $\big[P_1 \neq P_2 \textbf{ and } P_1 , P_2 \in \bigstar(\mathcal{E})\big] \textbf{ implies } P_1 \cap P_2 = \emptyset$

[8] Otherwise, if $p_1 , p_2 \notin V$ for all $V \in E$, the P-complete product represented by E contains $\overline{p}_1$ and $\overline{p}_2$ such that $\delta \overset{\text{sc}}{\not\models} p_1$ and $\delta \overset{\text{sc}}{\not\models} p_2$. This contradicts the assumption $|\{p_1 , p_2\} \cap \text{Dom}(\delta)| = 1$, which implies either $\delta \overset{\text{sc}}{\models} p_1$ or $\delta \overset{\text{sc}}{\models} p_2$.



(a) Merger$_3$ (b) Hourglass

Fig. 8: Maximal sets of indistinguishable ports of the hypergraphs in Figure 7

In summary, in the first step, transformation f_3 computes maximal sets of indistinguishable ports in every merged transition $t = (q\,,\,\psi\,,\,\phi\,,\,q')$ by applying $\bigstar$ to hyperedge relation $\mathcal{E}$ in hypergraph γ_t for ψ. Figure 8 shows examples.

In the second step, f_3 manipulates $\mathcal{E}$ of every hypergraph γ_t such that afterward, every vertex in every hyperedge in $\mathcal{E}$ is a part in $\bigstar(\mathcal{E})$. Importantly, every vertex $V \in E \in \mathcal{E}$ such that $V \in \bigstar(\mathcal{E})$ represents not just any $\bigoplus$-formula but one of indistinguishable ports. Consequently, in the meaning of the manipulated γ_t, indistinguishable ports become apparent as inner $\bigoplus$-formulas as in the example in the first paragraph of Section 5.

For manipulating hyperedge relation $\mathcal{E}$, we introduce an operation $\sqcup$ that combines two *combinable* hyperedges into one in a semantics-preserving way. Roughly, we call two distinct hyperedges $E_1\,,\,E_2 \in \mathcal{E}$ combinable if we can select disjoint vertices $V_1\,,\,V_2 \in E_1 \cup E_2$ such that E_1 and E_2 are equal up to inclusion of V_1 and V_2. We denote this property as $(E_1\,,\,V_1)\,\curlyvee_{\mathcal{E}}\,(E_2\,,\,V_2)$. Applied to combinable hyperedges E_1 and E_2, operation $\sqcup$ removes E_1 and E_2 from $\mathcal{E}$ and adds their combination $E_\dagger = \{V_1 \cup V_2\} \cup (E_1 \cap E_2)$ to $\mathcal{E}$. Formally, we have the following. Let $\mathbb{V}\mathrm{ER}$ denote the set of all vertices.

Definition 5. $\curlyvee \subseteq (\wp(\mathbb{V}\mathrm{ER}) \times \mathbb{V}\mathrm{ER}) \times (\wp(\mathbb{V}\mathrm{ER}) \times \mathbb{V}\mathrm{ER}) \times \wp^2(\mathbb{V}\mathrm{ER})$ *denotes the relation on tuples consisting of two sets of* $\big[$*hyperedge, vertex*$\big]$*-pairs and a set of hyperedges defined as:*

$$(E_1\,,\,V_1)\,\curlyvee_{\mathcal{E}}\,(E_2\,,\,V_2)\ \textbf{iff}\ \begin{bmatrix} E_1\,,\,E_2 \in \mathcal{E}\ \textbf{and}\ E_1 \neq E_2\ \textbf{and}\ V_1 \cap V_2 = \emptyset \\ \textbf{and}\ E_1 = (E_2 \setminus \{V_2\}) \cup \{V_1\} \\ \textbf{and}\ E_2 = (E_1 \setminus \{V_1\}) \cup \{V_2\} \end{bmatrix}$$

Definition 6. $\sqcup : (\wp(\mathbb{V}\mathrm{ER}) \times \mathbb{V}\mathrm{ER}) \times (\wp(\mathbb{V}\mathrm{ER}) \times \mathbb{V}\mathrm{ER}) \times \wp^2(\mathbb{V}\mathrm{ER}) \rightharpoonup \wp^2(\mathbb{V}\mathrm{ER})$ *denotes the partial function from tuples consisting of two* $\big[$*hyperedge, vertex*$\big]$*-pairs and a set of hyperedges to sets of hyperedges defined as:*

$$(E_1\,,\,V_1)\sqcup_{\mathcal{E}}(E_2\,,\,V_2) = \mathcal{E} \setminus \{E_1\,,\,E_2\}) \cup \{\{V_1 \cup V_2\} \cup (E_1 \cap E_2)\}$$
$$\textbf{if}\ \ (E_1\,,\,V_1)\,\curlyvee_{\mathcal{E}}\,(E_2\,,\,V_2)$$

Lemma 2. *(Lemma 8 in [11, Appendix A])*

$$(E_1\,,\,V_1)\,\curlyvee_{\mathcal{E}}\,(E_2\,,\,V_2)\ \textbf{implies}\ [\![(\mathcal{V}\,,\,\mathcal{E})]\!] \equiv_{\mathrm{sc}} [\![(\mathcal{V}\,,\,(E_1\,,\,V_1)\sqcup_{\mathcal{E}}(E_2\,,\,V_2))]\!]$$

Transformation f_3 uses operation $\sqcup$ in the algorithm for combining hyperedges in Figure 9. Essentially, as long as vertices V_1 and V_2 exist such that the

$$\textbf{while } \left[\begin{bmatrix}(X\,,\,V_1) \curlyvee_{\mathcal{E}} (Y\,,\,V_2) \textbf{ and}\\ V_1 \cup V_2 \subseteq P \textbf{ and } P \in \bigstar(\mathcal{E})\end{bmatrix} \textbf{ for some } X\,,\,Y\,,\,V_1\,,\,V_2\,,\,P\right] \textbf{ do}$$
$$\quad \textbf{while } \left[[(E_1\,,\,V_1) \curlyvee_{\mathcal{E}} (E_2\,,\,V_2)] \textbf{ for some } E_1\,,\,E_2\right] \textbf{ do}$$
$$\quad\quad \mathcal{E} := (E_1\,,\,V_1) \sqcup_{\mathcal{E}} (E_2\,,\,V_2)$$

Fig. 9: Algorithm for combining hyperedges

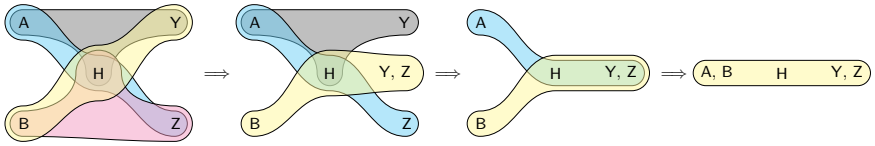


Fig. 10: Evolution of the hypergraphs in Figure 7b

ports in $V_1 \cup V_2$ are indistinguishable (as computed by $\bigstar$), the algorithm combines all combinable hyperedges that include V_1 and V_2. For instance, Figure 10 shows the evolution of the hypergraph in Figure 7b during the run of the algorithm in which it first selects Y and Z as V_1 and V_2 and afterward A and B. (In another run, the algorithm may change this order to obtain the same result.)

Let $\mathcal{E}_{\text{in}}$ and $\mathcal{E}_{\text{out}}$ denote the sets of hyperedges before and after running the algorithm. To consider the algorithm correct, $\mathcal{E}_{\text{out}}$ must satisfy two properties: it should represent an SC equivalent to the SC represented by $\mathcal{E}_{\text{in}}$ (i.e., the algorithm is semantics-preserving), and every vertex in every hyperedge in $\mathcal{E}_{\text{out}}$ should be a part in $\bigstar(\mathcal{E}_{\text{in}})$ (i.e., the algorithm effectively reveals indistinguishability). We use *Hoare logic* to prove these properties [14,15]. In particular, we can show that the triple $\{\mathsf{Pre}\}\ A\ \{\mathsf{Post}\}$ holds, where A denotes the algorithm in Figure 9. Precondition Pre states that $\gamma_t = (\mathcal{V}\,,\,\mathcal{E}_{\text{in}})$ is a hypergraph (for the SC of transition t) such that every port in a connected vertex inhabits at most one connected vertex, and such that every connected vertex is nonempty. The definition of graph in Definition 1 implies these conditions. (However, because its precondition is more liberal, the algorithm is more generally applicable.) The postcondition Post states that correctness as previously formulated holds. Formally:

$$[\![(\mathcal{V}\,,\,\mathcal{E}_{\text{out}})]\!] = [\![(\mathcal{V}\,,\,\mathcal{E}_{\text{in}})]\!] \textbf{ and } \left[\begin{bmatrix}E \in \mathcal{E}_{\text{out}} \textbf{ implies}\\ E \subseteq \bigstar(\mathcal{E}_{\text{in}})\end{bmatrix} \textbf{ for all } E\right]$$

Figure 11 shows the algorithm annotated with assertions for total correctness. By the axioms and rules of Hoare logic, this proof is valid if we can prove that for all six pairs of consecutive assertions, the upper assertion implies the lower one. For brevity, below, we discuss some salient aspects.

First, the algorithm terminates, because (i) every iteration of the outer loop consists of at least one iteration of the inner loop, for $X = E_1$ and $Y = E_2$, (ii) in every iteration of the inner loop, $\mathcal{E}$ decreases by one, and (iii) $\mathcal{E}$ is finite. Second, the algorithm is semantics-preserving by Lemma 2. The main challenge is proving that the algorithm is also effective. A notable step in this proof is establishing the property labeled Interm from Inv_2 (the invariant of the inner

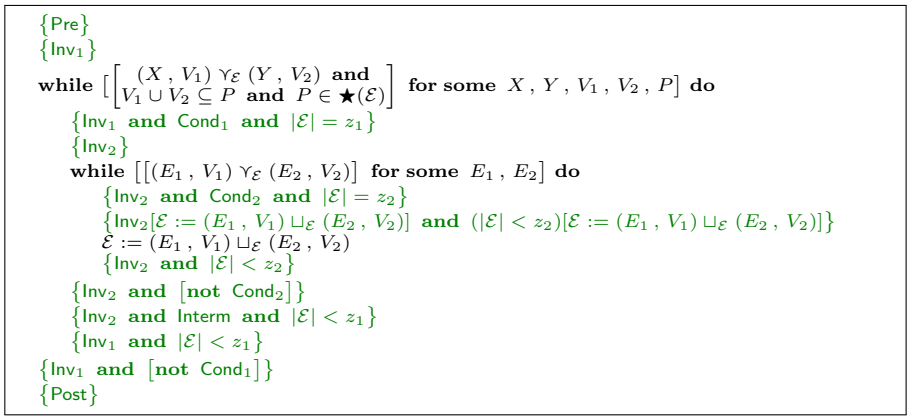


Fig. 11: Algorithm for combining hyperedges with assertions for total correctness

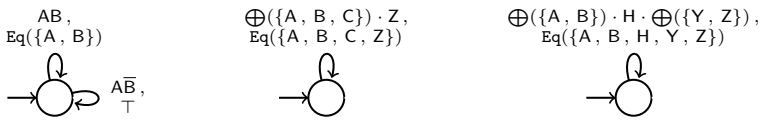
loop) and $[\textbf{not}\ \mathsf{Cond}_2]$ (the negation of the inner loop's condition). Informally, Interm states that if $\mathcal{F}$ denotes the hyperedge relation *before* running the inner loop, we have $\mathcal{E} = \mathcal{F} \setminus (\mathcal{F}_{1,2}) \cup \mathcal{F}_\dagger$ *after* running the inner loop. Here, $\mathcal{F}_{1,2}$ contains all hyperedges from $\mathcal{F}$ that include V_1 or V_2, while $\mathcal{F}_\dagger$ denotes all new hyperedges added by $\sqcup$ during the loop. This property subsequently enables us to prove Inv_1 (the invariant of the outer loop), which among other properties states $\bigstar(\mathcal{E}_{\text{in}}) = \bigstar(\mathcal{E})$. Consequently, to prove the algorithm's effectiveness, it suffices to show that $E \in \mathcal{E}_{\text{out}}$ implies $E \subseteq \bigstar(\mathcal{E}_{\text{out}})$ (for all E).

Theorem 2. *(Theorem 4 in [11, Appendix A])* $\{\mathsf{Pre}\}\ A\ \{\mathsf{Post}\}$

In summary, in the second step, for every (merged) transition $t = (q\,,\,\psi\,,\,\phi\,,\,q')$ in $f_1(\alpha)$, transformation f_3 manipulates hypergraph γ_t to γ_t' by running the algorithm in Figure 9, given the maximal sets of indistinguishable ports computed in f_3's first step with $\bigstar$. Afterward, f_3 replaces ψ in t with $[\![\gamma_t']\!]$, which by the correctness of the algorithm is equivalent to $[\![\gamma_t]\!]$ and has made indistinguishable ports (hence queues) apparent. We denote the resulting transition relation by $(f_3 \circ f_1)(\longrightarrow)$ and the resulting CA by $(f_3 \circ f_1)(\alpha)$. Because $\psi \equiv_{\text{sc}} [\![\gamma_t]\!] \equiv_{\text{sc}} [\![\gamma_t']\!]$ for all transitions t in $f_1(\alpha)$, the following proposition follows from Lemma 16 in [11, Appendix A]. Together, Propositions 1 and 2 imply that transformation f is semantics-preserving.

Proposition 2. $(f_3 \circ f_1)(\alpha) \sim f_1(\alpha)$

We end with some examples in Figure 12. Transformation f_3 has not had any effect on the LossySync CA, so its implementation does not benefit from queues (no indistinguishable ports), as expected. The Merger_3 and Hourglass CA, in contrast, have changed significantly. In the SC of Merger_3, we can now clearly recognize one queue for ports A, B, and C and one queue for port Z (cf. transformation h in Section 3); similarly, in the SC of Hourglass, we can now clearly recognize one queue for ports A and B and one queue for ports Y and Z.



(a) LossySync (b) Merger$_3$ (c) Hourglass

Fig. 12: Application of transformation f_3 to the CA in Figure 6

Applied to Merger$_3$, transformation f optimizes a multiple-producer-single-consumer protocol. More abstractly, in this case, f optimizes a protocol among two groups of processes, X_1 (producers) and X_2 (consumer), such that $|X_1| = 3$ and $|X_2| = 1$ and all processes in X_1 are indistinguishable to all processes in X_2 and vice versa. Generally, f can optimize protocols among n groups of processes $X_1 , \ldots , X_n$ such that for all $1 \leq i , j \leq n$, all processes in X_i are indistinguishable to all processes in X_j and vice versa. For instance, applied to Hourglass, f optimizes a protocol among three groups of processes such that $|X_1| = |X_3| = 2$ and $|X_2| = 1$.

After having applied transformation f, the automatic generation of actual implementations is straightforward (i.e., transformation g_2 in Figure 4c). The resulting code is, in fact, exactly the same as the code that results from manually applying transformation h as in Section 3 (and consequently, it has the same performance): instead of checking an `event` structure for every port as pre-optimized code does, optimized code checks one `eventQueue` structure for every maximal set of indistinguishable ports, which transformation f has made explicit as $\bigoplus$-formulas in SCs (and are thus easy to detect in the f-transformed CA). As such, optimized code checks the SC of all transitions in the pre-transformation CA that differ only in indistinguishable ports (before applying f) at the same time. For k such transitions, consequently, an unscalable exhaustive $\mathcal{O}(k)$ search is optimized to perfectly scalable $\mathcal{O}(1)$ queue operations. Thus, with respect to Figure 4c, the fully mechanical transformation $g_2 \circ f = g_2 \circ f_3 \circ f_2 \circ f_1$ yields the same code and scalability as the partially manual transformation $h \circ g_1$.

7 Concluding Remarks

In this paper, we analyzed scalability issues of the code generated by our Reo-to-C compiler, we explained a manual solution, and we studied the various steps of a mechanical procedure for transforming a CA α to an equivalent CA β, which makes port indistinguishability (hence queues) maximally apparent, using the $\bigoplus$-operator. Our tool can use this mechanical procedure to generate code for α via β with good scalability. In particular, whereas unoptimized code generated for α requires $\mathcal{O}(k)$ time to compute eligibility of k transitions—essentially an exhaustive search—the optimized code generated for β requires only $\mathcal{O}(1)$ time: all maximal sets of indistinguishable ports (explicit in β as a $\bigoplus$-formulas in SCs)

in the implementation share the same queue, which optimizes the unscalable $\mathcal{O}(k)$ search to perfectly scalable $\mathcal{O}(1)$ queue operations.

Although inspired by our work on a Reo compiler and formulated generally in terms of CA, we make contributions beyond Reo and CA. The synchronization pattern that we identified and optimized is common and occurs in many classes of protocols and their implementation, regardless of the particular language. Therefore, compilers for other high-level languages may use the same approach as explained in this paper to similarly optimize code generated for programs in those languages. In fact, this paper led to adding new features to Proto-Runtime to enable our optimization technique, thereby facilitating efficient implementation of our f-transformed CA. Importantly, these new features in Proto-Runtime can now benefit other languages implemented on top of Proto-Runtime as well.

Automatically performing our optimization directly on low-level code such as C (instead of on CA) is extremely complex, if not impossible. This shows that using higher-level languages (that preserve relevant context information about protocols) for multicore programming can indeed be advantageous for performance, a significant general observation in language and compiler design for multicore platforms. Indeed, the work presented in this paper serves as evidence that it is possible not only to specify interaction protocols at a higher level of abstraction (than locks, mutex, semaphores, message exchanges, etc.) but also automatically compile and optimize such high-level specifications down to executable code. Such higher-level specifications convey more of the intention behind the protocol, which gives more room for a compiler/optimizer to find and apply efficient implementation alternatives. Lower-level, more imperative, specifications of interaction protocols either lose or obscure the intentions behind protocols and seriously constrict the ability of compilers/optimizers to find efficient implementation alternatives. See [11] for related work on high-level approaches to multicore programming.

This paper makes primarily conceptual and theoretical contributions, and we used performance figures only to motivate and explain the development of our optimization technique. An in-depth study of the use of this technique in practice, including more benchmarks and experiments with different kinds of protocols and contexts, is our next objective, now that we know that the technique is correct. As part of this future work, we will also extend our current, limited proof-of-concept implementation (used in obtaining the data for Figure 5) to a full implementation. We end with the following remarks.

Indistinguishability of data. Transformation f effectively merges transitions with labels of the form $(\psi\,,\,\texttt{Eq}(P))$. The reason is that the ports in $\texttt{Eq}(P)$ are indistinguishable from a data perspective. (Whether those ports are also indistinguishable in ψ is exactly what transformation f_3 investigates.) Detecting port indistinguishability in *arbitrary* DCs so as to improve the applicability of f seems an interesting and important future challenge.

Guarded automata. Our SCs, as arbitrary propositional formulas, seem similar to guards on transitions in the guarded automata used by Bonsangue et al.

for modeling connector behavior [16]. The intuitive meaning of such guards, however, significantly differs: guards specify a constraint on the environment, while SCs specify a constraint on an execution step. (In fact, transition labels of guarded automata carry both a guard and an SC.)

Model-based testing. We skipped an explanation of the actual code generation process (i.e., transformation g_2 in Figure 4), dismissing it as "straightforward" and "obviously correct". An interesting line of work to better substantiate the latter statement is to have our tool generate not only executable code but also test cases derived from the input CA. Kokash et al. have already worked on such model-based testing for CA in a different context [17].

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. MSCS **14**(3) (2004) 329–366
2. Arbab, F.: Puff, The Magic Protocol. In: Talcott Festschrift. Volume 7000 of LNCS. Springer (2011) 169–206
3. Jongmans, S.S., Arbab, F.: Modularizing and Specifying Protocols among Threads. In: Proceedings of PLACES 2012. Volume 109 of EPTCS. CoRR (2013) 34–45
4. Jongmans, S.S., Arbab, F.: Overview of Thirty Semantic Formalisms for Reo. SACS **22**(1) (2012) 201–251
5. Kokash, N., Krause, C., de Vink, E.: Reo+mCRL2: A framework for model-checking dataflow in service compositions. FAC **24**(2) (2012) 187–216
6. Jongmans, S.S., Halle, S., Arbab, F.: Reo: A Dataflow Inspired Language for Multicore. In: Proceedings of DFM 2013. (2013)
7. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. SCP **61**(2) (2006) 75–113
8. Halle, S.: A Study of Frameworks for Collectively Meeting the Productivity, Portability, and Adoptability Goals for Parallel Software. PhD thesis, University of California, Santa Cruz (2011)
9. Halle, S., Cohen, A.: A Mutable Hardware Abstraction to Replace Threads. In: Proceedings of LCPC 2011. Volume 7146 of LNCS. Springer (2013) 185–202
10. Butenhof, D.: Programming with POSIX Threads. Addison-Wesley (1997)
11. Jongmans, S.S., Halle, S., Arbab, F.: Automata-based Optimization of Interaction Protocols for Scalable Multicore Platforms (Technical Report). Technical Report FM-1402, CWI (2014)
12. Sirjani, M., Jaghoori, M.M., Baier, C., Arbab, F.: Compositional Semantics of an Actor-Based Language Using Constraint Automata. In: Proceedings of COORDINATION 2006. Volume 4038 of LNCS. Springer (2006) 281–297
13. Bretto, A.: Hypergraph Theory: An Introduction. Springer (2013)
14. Hoare, T.: An Axiomatic Basis for Computer Programming. CACM **12**(10) (1969) 576–580
15. Apt, K., de Boer, F., Olderog, E.R.: Verification of Sequential and Concurrent Programs. Springer (2009)
16. Bonsangue, M., Clarke, D., Silva, A.: A model of context-dependent component connectors. SCP **77**(6) (2009) 685–706
17. Kokash, N., Arbab, F., Changizi, B., Makhnist, L.: Input-output Conformance Testing for Channel-based Service Connectors. In: Proceedings of PACO 2011. Volume 60 of EPTCS. CoRR (2011) 19–35