



Service Orchestration with Priority Constraints

Behnaz Changizi¹, Natallia Kokash^{2(✉)}, and Farhad Arbab³

¹ Leiden Institute of Advanced Computer Science,
Niels Bohrweg 1, Leiden, The Netherlands
behnaz.changizi@gmail.com

² Peoples' Friendship University of Russia (RUDN University),
6 Miklukho-Maklaya Street, Moscow 117198, Russian Federation
natallia.kokash@gmail.com

³ Centrum Wiskunde & Informatica, Science Park 123, Amsterdam, The Netherlands
farhad.arbab@cwi.nl

Abstract. Business process management is an operational management approach that focuses on improving business processes. Business processes, i.e., collections of important activities in an organization, are represented in the form of a workflow, an orchestrated and repeatable pattern of activities amenable to automated analysis and control. Priority is an important concept in modeling workflows. We need priority to model cancelable and compensable tasks within transactional business processes. We use the Reo coordination language to model and formally analyze workflows. In this paper, we propose a constraint-based approach to formalize priority in Reo. We introduce special channels to propagate and block priority flows, define their semantics as constraints, and model priority propagation as a constraint satisfaction problem.

Keywords: Transaction · Priority · Constraints · Coordination

1 Introduction

Business Process Management (BPM) systems [20, 23] are widely used to automate organizational business processes. Organizations rely on BPM to analyze, control or optimize their processes. BPM systems provide means for automated process analysis such as model validation, transformation, simulation, visualization of key performance indicators, and reporting [3]. Despite the variety of BPM systems [21, 29], The foundation of BPMN is based on Petri Nets [1, 32]. The choice of Petri Nets as foundation for BPMN implementation over other formal methods, often more expressive or specialized [13, 14], is not surprising: hardly any model is as simple, intuitive, and naturally supports task traceability.

While Petri net-based models enable automated process analysis, they lack a few desirable characteristics: (i) They cannot naturally represent semantics

of component-based or service-based processes. Ideally, we would like to plug semantic models for individual components (often integrated dynamically at run time) to the semantic models of existing processes in a compositional way. (ii) The classical Petri Nets are not expressive enough and often are extended (e.g., with colors, reset and inhibitor arcs, priority transitions) to enable meaningful process analysis. Such extensions change the semantics of the model and generate incompatible dialects of process-specification languages adopted by various tools.

An alternative formalization to express the semantics of BPMN models is the Reo coordination language [5]. Reo has been used to formalize semantics of BPMN, UML Activity and Sequence Diagrams [15], to map BPEL fragments [33], to represent transactional workflows [27], and to implement service orchestrations [24] and service choreographies [30]. Reo allows composition of components and services in an intuitive way, and addresses the issue (i) mentioned above. Moreover, the open-ended nature of Reo allows us to introduce channels with specific properties required for some applications. Introducing new primitives may make it necessary to extend the formal semantics of Reo in order to include some new concepts. Several dozen variations of semantic models for Reo have been proposed [25]. They vary from rather simple ones that cover basic Reo behavior (e.g., constraint automata [8]) to more complex models that cover specific behavioral aspects, e.g., context-sensitivity [18]. In some of these models, computing the overall semantics of a system is computationally expensive. This hampers using the language for analyzing large real-world business processes.

In [16], the authors proposed to model the semantics of Reo as a constraint satisfaction problem (CSP). They define data flow in a Reo network in the form of mathematical expressions on data observed at Reo nodes. The main advantage of such representation is the possibility to use existing constraint solvers to infer the behavior of a network given the semantics of its constituent parts.

Priority flow is an important aspect of process modeling, which is not easily supported by existing formalisms. Analyzing compensation and error handling requires a mechanism to express priority of some flow alternatives over others. In this paper, we propose a constraint-based framework for priority flow. There is ongoing work on an existing automata based formal semantics of Reo to handle priority, but our practical needs for dealing with large models of realistic business processes currently complicates direct use of automata-based semantic models.

This paper is organized as follows: In Sect. 2, we briefly describe the Reo coordination language. In Sect. 3, we introduce priority flow in Reo along with a constraint-based semantics for it. In Sect. 4, we extend our approach to support numeric priorities. In Sect. 5, we show the application of our constraint-based approach via two classes of connectors: (a) priority-aware, and (b) connectors with a large number of states. In Sect. 6, we overview related work. Finally, in Sect. 7, we conclude the paper and outline future work.

2 Reo

In the realm of service-oriented computing, the behavior of a software system is not only defined by the functionality of its services, but also by their interactions. The code written to realize the latter is often called *glue code*. Writing and maintaining glue code is a tedious task, especially in complex systems wherein the size and rigidity of the glue code tend to grow over time. Coordination languages offer a more manageable alternative for generating glue code. Reo [5,6] is a channel-based coordination language for composition of software components and services. Using a small and open-ended set of predefined and user-defined constructs, Reo supports modeling of complex coordination behavior.

The primitive constructs in Reo are *channels* and *nodes*, whose composition yields *connectors*. A channel is an atomic connector with two *ends* and a *constraint* that relates the flow of data at these ends. Channel ends are either *source* ends that read data into the channel or *sink* ends that write the channel's data out. Channels can connect to each other through nodes. There are two types of channel ends; therefore, three types of nodes can exist: *source nodes* where only source ends coincide, *sink nodes* where only sink ends coincide, and *mixed nodes* where both source and sink ends coincide. The mixed nodes of a connector are internal to the connector and not accessible for external data exchange. The source and sink nodes of a connector, collectively called its *boundary nodes* or *ports*, are used to connect to (the ports of) components to exchange data. A source node atomically replicates an incoming data items into all of its coincident channel ends, whenever they are all ready to accept. A sink node nondeterministically selects a data item out of one of its coincident channel ends and delivers it as its outgoing data item, leaving all other data items in its coincident channels intact. The behavior of a mixed node is an atomic combination of the behavior of a source node and that of a sink node: whenever all of its coincident source channels ends are ready to accept data items, it selects a data item out of one of its nondeterministically chosen coincident sink channel ends, and atomically replicates it into all of its source channel ends.

A *Sync* channel \longrightarrow has a source and a sink end. It accepts data from its source iff its sink can dispense it simultaneously. A *LossySync* \dashrightarrow has a source and a sink end. It reads a data item from its source and writes it simultaneously to its sink. If the sink end is not ready to accept the data item, the channel loses it. A *SyncDrain* \longleftrightarrow has two source ends and no sink end. It reads data from its two ends and discards it iff the ends are ready to interact simultaneously. A *FIFO₁* $\leftarrow\boxed{\rightarrow$ has a source end, a sink end, and capacity for only one data item. If it is empty, the channel accepts a data item from its source end and buffers it. If it is full, it is ready to dispense data through its sink end. Both ends of the channel cannot interact simultaneously. In addition to the primitive nodes, *Merger* and *Replicator*, here we use *Router* and *Cross-product*, which are shortcuts for derived connectors. The Reo nodes used in this work are explained as follows: A *Replicator* \multimap has one source end and one or more sink ends. It replicates data coming from its source to its sinks simultaneously. A *Merger* \multimap has one or more source ends and one sink end. It chooses one of its

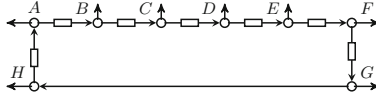


Fig. 1. 7-Sequencer

ready to interact source ends non-deterministically, receives a data item through this end, and writes it to its sink end simultaneously. A *Router* $\rightarrow \otimes$ has one source end and a number of sink ends. It accepts a data item from its source and simultaneously replicates it on one of its non-deterministically chosen sink, which is ready to accept data. A *Cross-product* $\otimes \rightarrow$ has a number of source ends and a sink end. It accepts a data item from each source, forms a tuple of them in the counter-clock-wise order with respect to its sink, where it writes the tuple, simultaneously.

3 Priority Flow

Here we define four channels to deal with priority in Reo.

A *PrioritySync* \leftrightarrow channel is similar to a *Sync* channel except it imposes priority on its flow, which propagates through the connector (unless it is blocked), and it can influence the non-deterministic choices in the containing connector by favoring data-flow alternatives that incorporate its ends. A *BlockSourceSync* channel \rightarrow is a *Sync* channel that blocks the propagation of priority from its source end towards its sink end. A *BlockSinkSync* channel \leftarrow is a *Sync* channel that stops propagation of priority from its sink end towards its source end. A *BlockSync* channel \leftrightarrow , a combination of *BlockSourceSync* and *BlockSinkSync*, stops the propagation of priority in both ways.

We model priority using the concepts of *innate* and *acquired* priority. Both ends of *priority sync* have *innate* priority. When an end with *innate* priority connects to another end that has no priority, the new end will obtain *acquired* priority. When one end of a synchronous type channel (e.g., *sync*, *lossy sync*, *sync drain*, ...) has *acquired* priority, the other end has *innate* priority.

However, in the case of non-synchronous channels (e.g., *FIFO*, *async drain*) and also the priority blocking channels, their ends can only have *acquired* priority. We update the constraint-based framework for Reo [16] to capture priority and the priority propagation mechanism, which we informally described above. In the rest of this paper, we omit data constraints when defining behavior of Reo elements. Data constraints are irrelevant for priority flow and were thoroughly covered in [16]. Motivated by the *constraint-based* nature of Reo itself, and the fact that constraint solving has advanced to the point that a number of practically useful constraint solvers exist today that can cope with realistically sized problems, we propose to define the behavior of Reo channels, as algebraic constraints that alter a set of variables.

Let \mathcal{N} and \mathcal{M} be global sets of ends and state memory variables, respectively. A free variable v has one of the following forms, where $n \in \mathcal{N}$ and $m \in \mathcal{M}$: $\tilde{n} \in \{\top, \perp\}$ shows presence or absence of data-flow on n ; $\dot{m}, \dot{m}' \in \{\top, \perp\}$ denotes whether or not the state memory variable m is defined in the source and the target states of the transition, respectively; $n^\triangleright \in \{\top, \perp\}$ indicates the reason for lack of data-flow on n originating from the primitive or the context (of this primitive), respectively; $n^{!^\bullet}, n^{!^\circ} \in \{\top, \perp\}$ models priority flow denoting whether n has *acquired* or *innate* priority. An end n has priority iff $n^{!^\bullet} \vee n^{!^\circ} = \top$.

A constraint Ψ , which encodes the behavior of a Reo network is defined as: $a:: = \tilde{n} \mid n^{!^\bullet} \mid n^{!^\circ} \mid n^\triangleright \mid \dot{m} \mid \dot{m}'$ (atoms), $\Psi:: = \top \mid a \mid \neg\Psi \mid \Psi \wedge \Psi$ (formulae) A solution to Ψ is a map from the variable sets V to a value in $\{\perp, \top\}$. The satisfaction rules for a solution $\langle \delta \rangle$ are satisfaction in propositional logic. We denote the set of all solutions for Ψ as $\mathfrak{S}(\Psi)$.

Definition 1 (RCSP). *A Reo Constraint Satisfaction Problem (RCSP) is a tuple $\langle \mathcal{N}, \mathcal{M}, M_0, \mathcal{V}, C \rangle$, where: \mathcal{N} is a finite set of ends. \mathcal{M} is a finite set of state memory variables. $M_0 \subseteq \mathcal{M}$ is a set of state memory variables that define the initial configuration of a network. \mathcal{V} is a set of variables v defined by the grammar $v:: = \tilde{n} \mid n^\triangleright \mid \dot{m} \mid \dot{m}' \mid n^{!^\circ} \mid n^{!^\bullet}$ for $n \in \mathcal{N}$ and $m \in \mathcal{M}$. $C = \{C_1, C_2, \dots, C_m\}$ is a finite set of constraints, where each C_i is a constraint given by the grammar Ψ involving a subset of variables $V_i \subseteq \mathcal{V}$.*

Definition 2. (Composition \odot). *The composition of two RCSPs $\rho_1 = \langle \mathcal{N}_1, \mathcal{M}_1, M_{0,1}, \mathcal{V}_1, C_1 \rangle$ and $\rho_2 = \langle \mathcal{N}_2, \mathcal{M}_2, M_{0,2}, \mathcal{V}_2, C_2 \rangle$ is defined as follows: $\rho_1 \odot \rho_2 = \langle \mathcal{N}_1 \cup \mathcal{N}_2, \mathcal{M}_1 \cup \mathcal{M}_2, M_{0,1} \cup M_{0,2}, \mathcal{V}_1 \cup \mathcal{V}_2, C_1 \wedge C_2 \rangle$.*

Axiom 1 (Join axiom). *To propagate no-flow reasons, when a source end c and a sink end k from two networks, the following holds: $\neg\tilde{c} \Leftrightarrow \neg\tilde{k} \Leftrightarrow (c^\triangleright \vee k^\triangleright)$.*

Axiom 2 (Priority join axiom). *When a source end c and a sink end k from two networks connect, this holds: $(c^{!^\circ} \vee c^{!^\bullet} \Leftrightarrow k^{!^\circ} \vee k^{!^\bullet}) \wedge (c^{!^\circ} \wedge k^{!^\circ} \Leftrightarrow c^{!^\bullet} \vee k^{!^\bullet})$.*

Axiom 3 (Non-deterministic choice axiom). *Let N be a set of ends from which a Reo primitive chooses one for communication non-deterministically. The following guarantees that a node y with no priority has flow only if no prioritized node, e.g., x , is ready to interact: $(\neg\tilde{x} \wedge (x^{!^\circ} \vee x^{!^\bullet})) \wedge \tilde{y} \wedge \neg(y^{!^\circ} \vee y^{!^\bullet}) \Rightarrow \neg x^\triangleright$*

In [16], the authors described the constraints that a primitive imposes on a network as a CSP. We extend these constraints with priority capturing variables. If the variable $p^{!^\bullet}$ is *true*, the end p has *innate* priority. For example, in a *PrioritySync* channel, both ends have *innate* priority. A primitive end can also obtain *innate* priority via propagation. For instance, if one end of a *Sync* channel has *acquired* priority, which means it is prioritized because a primitive connected to it propagates priority, then the other end will have *innate* priority. We denote *acquired* priority for a primitive end p as: $p^{!^\circ} \wedge \neg p^{!^\bullet}$. The priority capturing constraint for a *Sync* channel with source end a and sink end b can be specified as follows: $\neg(a^{!^\circ} \vee a^{!^\bullet} \vee b^{!^\circ} \vee b^{!^\bullet}) \vee (a^{!^\circ} \wedge \neg a^{!^\bullet} \wedge b^{!^\bullet}) \vee (a^{!^\bullet} \wedge b^{!^\circ} \wedge \neg b^{!^\bullet})$.

```

1 Input: A Reo network  $R$  and its RCSP  $\psi$ , Output: Solutions for the given RCSP
2  $\text{fifoStates} \leftarrow$  initial states of FIFOs from the given RCSP;
3  $\text{state}_0 \leftarrow \{\langle \text{fifoStates} \rangle\}$ ;  $\text{toExplore} \leftarrow \{\text{state}_0\}$ ;  $\text{visited} \leftarrow \{\}$ ;  $\text{solutions} \leftarrow \{\}$ ;
4 while ( $\text{toExplore} \neq \{\}$ ) do
5    $\text{state} \leftarrow \text{toExplore.pop}()$ ;  $\text{visited} \leftarrow \text{visited} \cup \{\text{state}\}$ ;
6    $\text{cnf} \leftarrow \text{updateStateAndMakeCNF}(\psi, \text{state})$ ;
7    $\text{solutions}_B \leftarrow \text{solve}(\text{cnf})$ ;
8   for  $\text{sol}_B \in \text{solutions}_B$  do
9      $\text{state}' \leftarrow$  next state of FIFOs extracted from  $\text{sol}_B$ ;
10    if  $\text{state}' \notin \text{visited}$  and  $\text{state}' \notin \text{toExplore}$  then
11       $\text{toExplore} \leftarrow \text{toExplore} \cup \{\text{state}'\}$ ;
12    end
13     $\text{solutions} \leftarrow \text{solutions} \cup \{\langle \text{state}, \text{sol}_B, \text{state}' \rangle\}$ ;
14  end
15   $\text{output} \leftarrow \{\text{solutions}, \text{state}_0\}$ ;
16 end

```

Algorithm 1. Finding solutions for a given RCSP

The assertion $\neg p^{i^\bullet}$ blocks the priority propagation on p . Though, p can still have *acquired* priority through a potential connecting primitive when $p^{i^\circ} = \top$.

Table 1 shows the constraint encoding of Reo channels and nodes in presence of priority flow. The solutions to the CSP expressing the behavior of a Reo network encode possible data-flow through its nodes. Since a network may later connect to another network, the constraints should account for priority imposed by potential future connections. This information can be discarded when analyzing the behavior of a network in isolation. To exclude such cases, we should restrict the possible values of boundary ends.

Axiom 4 (Grounding axiom). *Let $B \subset N$ be the set of boundary nodes in a Reo network. We rule out the solutions that are only present for further expansion of the network by: $\forall b \in B : b^{i^\circ} \Rightarrow b^{i^\bullet}$.*

Solutions of the RCSP represent semantics of the corresponding Reo network, but they are specified as equations, which are much harder to interpret than an equivalent automata-based semantics. To tackle this issue, we introduce a new form of automata-like semantics for Reo, which we call Reo Labeled Transition System (RLTS). The purpose of the RLTS is to compactly represent solutions of RCSPs for visualization, model checking and simulation. Given a Reo network, its RCSP can be obtained by traversing the network and forming the conjunction the constraint encodings of its primitives. The procedure to solve an RCSP is presented in Algorithm 1. It takes a Reo connector and its RCSP and outputs the solutions set and the initial state of the connector. First, the algorithm initializes the global variables that keep the states of FIFO channels (fifoStates), the states to explore (toExplore), and the visited states (visited) (lines 2,3). While toExplore is not empty, Ψ is updated with the current state and its conjunctive normal form (CNF) is produced for computing the solutions of the Boolean predicates (lines 4,5). The $\langle \text{state}' \rangle$ indicates the new state of the connector and if it is not already explored or queued to be processed, it gets added to the list of states to be explored (lines 6–9). Then, the solutions set is updated with the current solution (line 13). The final output is the set of solutions and the initial state.

Table 1. Constraint encoding of Reo with priority

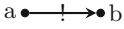

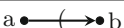



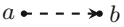

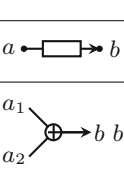
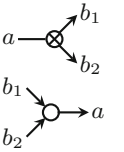
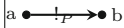





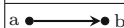

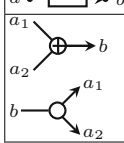
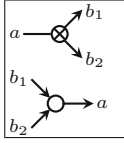
Channel	Constraints
	$(\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^\triangleright \wedge b^\triangleright) \wedge a^{!^\bullet} \wedge b^{!^\bullet}$
	$(\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^\triangleright \wedge b^\triangleright) \wedge \neg b^{!^\bullet}$
	$(\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^\triangleright \wedge b^\triangleright) \wedge \neg a^{!^\bullet}$
	$(\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^\triangleright \wedge b^\triangleright) \wedge \neg a^{!^\bullet} \wedge \neg b^{!^\bullet}$
	$(\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^\triangleright \wedge b^\triangleright) \wedge ((\neg a^{!^\bullet} \wedge \neg a^{!^\circ} \wedge \neg b^{!^\bullet} \wedge \neg b^{!^\circ}) \vee ((a^{!^\circ} \Rightarrow b^{!^\bullet}) \wedge (b^{!^\circ} \Rightarrow a^{!^\bullet})))$
	$(\tilde{b} \Rightarrow \tilde{a}) \wedge \neg a^\triangleright \wedge \neg \tilde{a} \Rightarrow b^\triangleright \wedge ((\neg a^{!^\bullet} \wedge \neg a^{!^\circ} \wedge \neg b^{!^\bullet} \wedge \neg b^{!^\circ}) \vee ((a^{!^\circ} \Rightarrow b^{!^\bullet}) \wedge (b^{!^\circ} \Rightarrow a^{!^\bullet})))$
	$(\tilde{a}_1 \Leftrightarrow \tilde{a}_2) \wedge \neg(a_1^\triangleright \wedge a_2^\triangleright) \wedge ((\neg a^{!^\bullet} \wedge \neg a^{!^\circ} \wedge \neg b^{!^\bullet} \wedge \neg b^{!^\circ}) \vee (a^{!^\circ} \Rightarrow b^{!^\bullet}) \wedge (b^{!^\circ} \Rightarrow a^{!^\bullet}))$
	$(\tilde{a} \Rightarrow \neg \hat{m} \wedge \hat{m}') \wedge (\tilde{b} \Rightarrow \hat{m} \wedge \neg \hat{m}') \wedge (\neg \tilde{a} \wedge \neg \tilde{b}) \Rightarrow (\hat{m} \Leftrightarrow \hat{m}') \wedge (\neg \hat{m} \Rightarrow b^\triangleright) \wedge (\hat{m} \Rightarrow a^\triangleright) \wedge (\neg a^{!^\bullet} \wedge \neg b^{!^\bullet})$
	$\tilde{a} \Leftrightarrow (\tilde{b}_1 \wedge \tilde{b}_2) \wedge \neg \tilde{a} \Rightarrow ((\neg a^\triangleright \wedge b_1^\triangleright \wedge b_2^\triangleright) \vee (\neg b_1^\triangleright \wedge b_2^\triangleright \wedge a^\triangleright) \vee (\neg b_2^\triangleright \wedge b_1^\triangleright \wedge a^\triangleright)) \wedge ((\neg a^{!^\bullet} \wedge \neg b_1^{!^\bullet} \wedge \neg b_2^{!^\bullet} \wedge \neg a^{!^\circ} \wedge \neg b_1^{!^\circ} \wedge \neg b_2^{!^\circ}) \vee ((a^{!^\circ} \Rightarrow (b_1^{!^\bullet} \wedge b_2^{!^\bullet})) \wedge (b_1^{!^\circ} \vee b_2^{!^\circ}) \Rightarrow a^{!^\bullet}))$
	$\tilde{a} \Leftrightarrow (\tilde{b}_1 \vee \tilde{b}_2) \wedge \neg(\tilde{b}_1 \wedge \tilde{b}_2) \wedge \tilde{a} \Leftrightarrow (\neg a^\triangleright \vee \neg(b_1^\triangleright \vee b_2^\triangleright)) \wedge ((\neg a^{!^\bullet} \wedge \neg b_1^{!^\bullet} \wedge \neg b_2^{!^\bullet} \wedge \neg a^{!^\circ} \wedge \neg b_1^{!^\circ} \wedge \neg b_2^{!^\circ}) \vee (\tilde{a} \Rightarrow ((a^{!^\circ} \Rightarrow (b_1^{!^\bullet} \vee b_2^{!^\bullet})) \wedge (b_1^{!^\circ} \vee b_2^{!^\circ}) \Rightarrow a^{!^\bullet}) \wedge (\tilde{b}_1 \Rightarrow (a^{!^\circ} \Rightarrow b_1^{!^\bullet} \wedge b_1^{!^\circ} \Rightarrow a^{!^\bullet}) \wedge ((\neg b_1^{!^\circ} \wedge \neg b_1^{!^\bullet} \wedge \neg \tilde{b}_2 \wedge (b_2^{!^\circ} \vee b_2^{!^\bullet})) \Rightarrow \neg b_2^\triangleright) \wedge (\tilde{b}_2 \Rightarrow (a^{!^\circ} \Rightarrow b_2^{!^\bullet} \wedge b_2^{!^\circ} \Rightarrow a^{!^\bullet} \wedge (\neg b_2^{!^\circ} \wedge \neg b_2^{!^\bullet} \wedge \neg \tilde{b}_1 \wedge (b_1^{!^\circ} \vee b_1^{!^\bullet}) \Rightarrow \neg b_1^\triangleright))))))$

Table 2. Updating Priority capturing constraints

				
$a^{!^\bullet} \geq P \wedge b^{!^\bullet} \geq P$	$b^{!^\bullet} = 0$	$a^{!^\bullet} = 0$	$a^{!^\bullet} = 0 \wedge b^{!^\bullet} = 0$	$a^{!^\bullet} = 0 \wedge b^{!^\bullet} = 0$
	$(a^{!^\bullet} = 0 \wedge a^{!^\circ} = 0 \wedge b^{!^\bullet} = 0 \wedge b^{!^\circ} = 0) \vee ((a^{!^\circ} > 0 \Rightarrow (a^{!^\circ} = b_1^{!^\bullet})) \wedge (b_1^{!^\circ} > 0 \Rightarrow (b_1^{!^\circ} = a^{!^\bullet}))) \wedge (b^{!^\bullet} > 0 \Rightarrow \tilde{b})$			
	$(a^{!^\bullet} = 0 \wedge a^{!^\circ} = 0 \wedge b^{!^\bullet} = 0 \wedge b^{!^\circ} = 0) \vee ((a^{!^\circ} > 0 \Rightarrow (a^{!^\circ} = b^{!^\bullet})) \wedge (b^{!^\circ} > 0 \Rightarrow (b^{!^\circ} = a^{!^\bullet})))$			
	$a^{!^\bullet} = 0 \wedge b^{!^\bullet} = 0$			
	$((a^{!^\bullet} = 0 \wedge a^{!^\circ} = 0 \wedge b^{!^\bullet} = 0 \wedge b^{!^\circ} = 0) \vee ((b_1^{!^\circ} > 0 \Rightarrow (a_1^{!^\bullet} = b_1^{!^\circ} \wedge a_2^{!^\bullet} = b_1^{!^\circ})) \wedge (a_1^{!^\circ} > 0 \Rightarrow (a_2^{!^\bullet} = a_1^{!^\circ} \wedge b^{!^\bullet} = a_1^{!^\circ}) \wedge (a_2^{!^\circ} > 0 \Rightarrow (a_1^{!^\bullet} = a_2^{!^\circ} \wedge b^{!^\bullet} = a_2^{!^\circ}))))))$			
	$((a^{!^\bullet} = 0 \wedge a^{!^\circ} = 0 \wedge b^{!^\bullet} = 0 \wedge b^{!^\circ} = 0) \vee (\tilde{b}_1 \Rightarrow (b_1^{!^\circ} > 0 \Rightarrow (b_1^{!^\circ} = a^{!^\bullet}))) \wedge (\tilde{b}_2 \Rightarrow (b_2^{!^\circ} > 0 \Rightarrow (b_2^{!^\circ} = a^{!^\bullet}))) \wedge ((\max(b_1^{!^\circ}, b_1^{!^\bullet}) > \max(b_2^{!^\circ}, b_2^{!^\bullet})) \Rightarrow ((\tilde{b}_2 \wedge \neg \tilde{b}_1) \Rightarrow \neg b_2^\triangleright) \wedge ((\max(b_2^{!^\circ}, b_2^{!^\bullet}) > \max(b_1^{!^\circ}, b_1^{!^\bullet})) \Rightarrow ((\tilde{b}_1 \wedge \neg \tilde{b}_2) \Rightarrow \neg b_2^\triangleright)))$			

Definition 3 (RLTS). A *Reo Labeled Transition System (RLTS)* is a tuple $\mathcal{RLTS}=(\mathcal{N}, \mathcal{M}, Q, \rightarrow, q_0)$, where: \mathcal{N} is a set of ends, \mathcal{M} is a set of state memory variables, Q is a (finite) set of states of the form $\langle M \rangle$, M is the set of state memory variables that are valid in the given state, $\rightarrow \subseteq Q \times 2^{\mathcal{N}} \times 2^{\mathcal{N}} \times 2^{\mathcal{N}} \times Q$ is a transition relation, wherein $N, R,$ and I in $(q, N, R, I, p) \in \rightarrow$ represent the ends that have flow, those without flow for which the reason for no flow is the end not being ready for interaction, and the ends with priority. Note that $n \notin N$ does not always mean $n \in R$ as the reason for data flow can be the network (then, n requires a reason for no flow). $q_0 \in Q$ is the initial state. We write $q \xrightarrow{N, R, I} p$ instead of $(q, N, R, I, p) \in \rightarrow$. For $n \in I, n \notin R \Leftrightarrow n \in N$.

Definition 4 (Composition \square). We define the composition of $L_1 = (\mathfrak{N}_1, \mathcal{M}_1, Q_1, \rightarrow_1, q_{0_1})$ and $L_2 = (\mathfrak{N}_2, \mathcal{M}_2, Q_2, \rightarrow_2, q_{0_2})$ as: $L_1 \square L_2 = (\mathcal{N}_1 \cup \mathcal{N}_2, \mathcal{M}_1 \cup \mathcal{M}_2, \rightarrow, q_{0_1} \times q_{0_2})$ where \rightarrow is defined as:

$$q_1 \xrightarrow{N_1, R_1, I_1} t_1 t_2 \xrightarrow{N_2, R_2, I_2} 2t_2 N_1 \cap \mathfrak{N}_2 = N_2 \cap \mathfrak{N}_1 R_1 \cap \mathfrak{N}_2 = R_2 \cap \mathfrak{N}_1 I_1 \cap \mathfrak{N}_2 = I_2 \cap \mathfrak{N}_1 \\ q_1 \times q_2 \xrightarrow{N_1 \cup N_2, R_1 \cup R_2, I_1 \cup I_2} t_1 \times t_2$$

$$q_1 \xrightarrow{N_1, R_1, I_1} t_1 t_2 \xrightarrow{N_2, R_2, I_2} 2t_2 N_1 \cap \mathfrak{N}_2 = \emptyset, \text{ and its symmetric rule.} \\ q_1 \times q_2 \xrightarrow{N_1, R_1, I_1} t_1 \times t_2$$

We define few operations on a solution s for $\Psi = \langle \mathcal{N}_\Psi, \mathcal{M}_\Psi, M_{\Psi 0}, \mathcal{V}_\Psi, C_\Psi \rangle$: $\text{source}(s) = \{m | m^\circ \in \mathcal{M}_\Psi : s(m^\circ) = \top\}$, $\text{target}(s) = \{m | m'^\circ \in \mathcal{M}_\Psi : s(m'^\circ) = \top\}$, $\text{flow}(s) = \{n | n \in \mathcal{N}_\Psi : s(\tilde{n}) = \top\}$, $\text{reason-giving}(s) = \{n | n \in \mathcal{N}_\Psi : s(n^\triangleright) = \top\}$, $\text{priority}(s) = \{n | n \in \mathcal{N}_\Psi : (s(n^\circ) \vee s(n^\bullet)) = \top\}$. We say $s \smile q \xrightarrow{N, R, I} p$, where $q = \text{source}(s)$, $N = \text{flow}(s)$, $R = \text{reason-giving}(s)$, $I = \text{priority}(s)$, $p = \text{target}(s)$.

Definition 5 (Visualization). The visualization function γ on $\Psi = \langle \mathcal{N}, \mathcal{M}, M_0, \mathcal{V}, C \rangle$ yields $\mathcal{L} = (\mathcal{N}, \mathcal{M}, Q, \rightarrow, q_0)$, where $\mathcal{M} = \{m | s(m^\circ) = \top \vee s(m'^\circ) = \top, s \in \mathfrak{S}(\Psi)\}$, $Q = \bigcup_{s \in \mathfrak{S}(\Psi)} \{\text{source}(s), \text{target}(s)\}$, $\rightarrow = \{(\text{source}(s), \text{flow}(s), \text{reason-giving}(s), \text{priority}(s), \text{target}(s)) | s \in \mathfrak{S}(\Psi)\}$, $q_0 = \text{source}(s_0)$.

Theorem 1. Let Ψ_1 and Ψ_2 be two RCSPs, we show that $\gamma(\Psi_1 \odot \Psi_2) = \gamma(\Psi_1) \square \gamma(\Psi_2)$.

Proof. Let $\gamma(\Psi_1) = (\mathfrak{N}_1, \mathcal{M}_1, Q_1, \rightarrow_1, q_{0_1})$, $\gamma(\Psi_2) = (\mathfrak{N}_2, \mathcal{M}_2, Q_2, \rightarrow_2, q_{0_2})$, and $\gamma(\Psi_1 \odot \Psi_2) = (\mathfrak{N}, \mathcal{M}, Q, \rightarrow, q_0)$. It is trivial to see that $\mathfrak{N} = \mathfrak{N}_1 \cup \mathfrak{N}_2$, $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$, $Q = Q_1 \times Q_2$, $q_0 = q_{0_1} \times q_{0_2}$. Assume $\exists s \in \mathfrak{S}(\Psi_1 \odot \Psi_2)$, $s_1 \in \mathfrak{S}_1$, $s_2 \in \mathfrak{S}_2$, $t_1 : q_1 \xrightarrow{N_1, R_1, I_1} 1p_1$, $t_2 : q_2 \xrightarrow{N_2, R_2, I_2} 2p_2$ s.t. $s_1 \smile t_1$ and $s_2 \smile t_2$, but $\nexists t : q \xrightarrow{N, R, I} p \in \rightarrow$ s.t. $s \smile t$. Therefore, $N_1 \cap \mathfrak{N}_2 \neq N_2 \cap \mathfrak{N}_1 \wedge N_1 \cap \mathfrak{N}_2 \neq \emptyset$ or $(N_1 \cup N_2) \cap (R_1 \cup R_2) \neq \emptyset$. The latter is impossible. For the former, either $n \in N_1, n \notin N_2$ or $n \in N_2, n \notin N_1$, which is not possible as it means $s(n) = \top \wedge s(n) = \perp$. Similarly, we can show it is impossible to have a t in $\gamma(\Psi_1 \odot \Psi_2)$, when there is no $s \in \mathfrak{S}$ s.t. $s \smile t$.

RLTS is comparable with *Reo automata* [12], a context-dependent formal semantic of Reo. A transition in *Reo automata* is labeled with a *guard*, which is a

Boolean predicate in disjunctive normal form expressing positive and negative information about presence or absence of I/O requests, and a *firing* set that models the occurring I/O operations in the transition. The second set in RLTS transitions (the set of ends that provide reason for no flow) correspond to the negated elements of the guards in *Reo automata*, while the set of ends with flow relates to both the *firing* set and the positive elements of the guards. Unlike *Reo automata*, RLTS supports priority.

4 Numeric Priority

In BPMN, an *error* event has the highest priority, and the *exception* has priority over the normal flow. In this extension, the range for priority variables of an end n , $n^{!^\circ}$ and $n^{!^\bullet}$, is \mathbb{N} (natural numbers) $\cup \{0\}$, where 0 indicates no priority. The larger number is the higher priority it represents. Each *PrioritySync* channel comes with a user defined priority value, which propagates through its ends. To propagation of a higher priority over a lower priority or no priority, we constrain priority variables to be greater than or equal to their initial values. Table 2 shows the priority related parts of the Reo constructs constraints. $\langle \delta \rangle \models x \geq P$ iff $\delta(x) \geq P$, $\langle \delta \rangle \models x > P$ iff $\delta(x) > P$, $\langle \delta \rangle \models x = P$ iff $\delta(x) = P$, where $x \in \{x^{!^\bullet}, x^{!^\circ}\}$, $P \in \mathbb{N} \cup \{0\}$. The new constraint-based encodings of the *replicator* and *router* nodes in this table are constructed in accordance with Axiom 3.

Definition 6 (NPRLTS). A *Numeric Priority Reo Labeled Transition System* is a tuple $(\mathcal{N}, \mathcal{M}, Q, \rightarrow, q_0)$, where: \mathcal{N} is a set of ends, \mathcal{M} is a set of state memory variables, Q is a (finite) set of states of the form $\langle M \rangle$, M is the set of state memory variables that are valid in the given state, $\rightarrow \subseteq Q \times 2^{\mathcal{N}} \times 2^{\mathcal{N}} \times \mathcal{N} \mapsto \mathbb{N} \times Q$ is a transition relation, wherein N, R , and f_I in $(q, N, R, f_I, p) \in \rightarrow$ are the ends having flow, those without flow for which the reason for no flow is the end not being ready for interaction, and a partial map of nodes with priority to their priority values, respectively. $q_0 \in Q$ is the initial state. We write $q \xrightarrow{N, R, f_I} p$ instead of $(q, N, R, f_I, p) \in \rightarrow$. For all $q \xrightarrow{N, R, f_I} p$: $f(n) > 0, n \notin N \Leftrightarrow n \in R$. We redefine *priority*(s) as $\{(n, p) | n \in \mathcal{N}_\Psi : s(n^{!^\circ}) = p \vee s(n^{!^\bullet}) = p\}$.

Definition 7 (Extended Visualization). The visualization function γ on $\Psi = \langle \mathcal{N}_\Psi, \mathcal{M}_\Psi, M_{\Psi_0}, \mathcal{V}, C \rangle$ yields $\mathcal{L} = (\mathcal{N}_L, \mathcal{M}_L, Q, \rightarrow, q_0)$, where $\mathcal{N}_L = \{n | s(\tilde{n}) = \top, s \in \mathfrak{S}(\Psi)\}$, $\mathcal{M}_L = \{m | s(m^\circ) = \top \vee s(m'^\circ) = \top, s \in \mathfrak{S}(\Psi)\}$, $Q = \bigcup_{s \in \mathfrak{S}(\Psi)} \{source(s), target(s)\}$, $\rightarrow = \{(source(s), flow(s), reason-giving(s), priority(s), target(s)) \mid s \in \mathfrak{S}(\Psi)\}$, $q_0 = source(s_0)$.

5 Case Study

Here we demonstrate the application of our approach via an example and present a performance evaluation of our approach. Figure 2(a) depicts a sales process, which starts by receiving an order. It proceeds by reserving the ordered items

for the customer. Then, the customer gets charged and her account is updated. Meanwhile if the payment encounters a problem, a *cancellation* event is triggered, which causes compensation for all of the performed activities. However, if an *error* event occurs, all tasks inside the transaction stop, the *boundary error catch* event redirects the flow to notifying the operator. Finally, if no problem occurs, the ordered items are shipped and the process ends.

Figure 2(c) shows a Reo network that simulates this process. The process starts by reading a token from the *writer* W_2 , which resembles receiving an order. Though a Reo network can be used for modeling infinite data flow, in the BPMN standard, when a *start* event is triggered, a new instance of the process is instantiated. Therefore, the Reo network is designed to handle only one request. The end A_1 reads a token from the writer W_2 and directs it to *replicator* node B , which duplicates the token and forwards them to the BC and BE $FIFO_1$ channels. The token from BC continues to the CD $FIFO_1$ channel. If the payment succeeds, the flow from CD and BE $FIFO_1$ channels merge and a token enters the FG $FIFO_1$ channel. Then, it gets consumed by the *reader* R_3 .

If the payment fails, performed activities need to be compensated. A token from W_1 simulates a payment failure, so the process needs to be canceled. The *prioritySync* channel IJ imposes a priority of *one* on the failure associated flow. The node J replicates the failure token into the *lossySync* channels JM and JU , depending on whether each of the $FIFO_1$ channels BC and CD is empty or full, the connected *lossySync* channels lose the incoming tokens or pass them to the adjacent *syncDrain* to consume the tokens of $FIFO_1$ channels, respectively. At the same time, the *replicator* node J writes into the $FIFO_1$ channels JK and JN , which simulate *cancel reservation* and *undo changes* tasks, respectively. The flow corresponding to *error*, starting from the *writer* W_3 , is structurally similar to the failure flow, but it has a priority of 2 due to SQ *PrioritySync*.

To analyze the presented BPMN process, we convert it to a Reo network. The core mapping is presented in [7, 17], which maps a *task* to a $FIFO_1$ channel, while it converts *message*, *cancel*, and *error* events to *writer* components simulating the incoming flows from the environment. A diverging *parallel* gateway is mapped to a *replicator*, while a converging *parallel* gateway is mapped to a *join*. The *sequence flows* are converted into *sync* channels. The mapping of *exception* and *error* handling flows are more complex and are presented in [27].

In this example, the *error* handling flow has the highest priority, while the *exception* handling has the medium priority, and the *success* flow has no priority. The choice between these three alternative flows is made by the *routers*. We obtain the NPRLTS as follows: First, we form the RCSP of the network by traversing through its primitives. Then, we solve the obtained RCSP and extract transitions from obtained solutions, as described in Algorithm 1.

To show the effect of priority on our example, we first investigate the behavior of the network in absence of priority, wherein the normal flow of the process can continue even in case of a payment failure. This is because the *router* node E chooses one of its outgoing flows non-deterministically. The following assets a priority-respecting routing of these alternative flows. $(\{BE\} \in source(t) \wedge (C_1 \in$

$flow(t) \vee E_1 \in flow(t)) \Rightarrow ((W_3 \notin reason-giving(t) \Leftrightarrow W_3 \in flow(t)) \wedge (W_3 \in reason-giving(t) \wedge W_1 \notin reason-giving(t)) \Leftrightarrow W_1 \in flow(t))$.

A typical way of verifying this property is to check it against the NPRLTS of the network. The given property is straight forward to check. Due to the number of ends in this example, the transition labels of the NPRLTS are lengthy. Thus for brevity, we apply an abstraction on the original NPRLTS, which leads to a more concise and readable model. To address a node end, we append a number index to the node name (e.g., B_1). We refer to a channel using the name of the nodes connected to its ends (e.g., BC). Similarly, we append a number index to a channel name to denote a channel end (e.g., BC_1). In addition, we group the ends with a similar name e.g., $B_{1,2}$ (referring to ends B_1 and B_2).

Since, the property solely mentions the ends C_1 , E_1 , W_1 , and W_3 on the transitions originating from the states where BE $FIFO_1$ channel is full, we abstract from the rest of the ends in those transitions and from all the ends in other transitions. It is straight-forward to see that this abstraction does not affect the correctness of the validation due to the nature of the property.

Figure 2(b) shows the abstract NPRLTS of the network of Fig. 2(c) in absence of priority. The property that we are interested to check is that if from any state wherein BE holds, W_3 has flow unless it provides a reason for no flow itself, and if W_3 provides a reason for no flow, W_1 has flow unless it provides a reason for no flow itself. This property, however, does not hold on the current NPRLTS as it contains transitions originating from states $\{BC, BE\}$ and $\{CD, BE\}$, wherein either W_3 is absent in R (the set of ends providing a reason for now flow), yet it is not in N (the set of ends with data flow) or W_3 is in R , but W_1 is not in R , yet it is not in N .

Here we show how considering priority constraints rules out these transitions. We reason about one of the transitions (the transition from $\{BC, BE\}$ to $\{CD, BE\}$ with $N = \{C_1, \dots\}$, $R = \{W_1, \dots\}$). Similar reasonings hold for the rest.

$$\begin{aligned}
0 &: \frac{\exists t \in NPRLTS: C_1 \in N(t), W_1 \in R(t), W_3 \notin N(t), W_3 \notin R(t)}{\exists s \in \mathfrak{S}(\Psi) \text{ s.t. } s \Rightarrow \bar{C}_1 \wedge \neg W_3 \wedge W_1^\flat \wedge \neg W_3^\flat} \\
1 &: \frac{0 \& \Psi_{PrioritySync_1}(SQ_{1,2}) \& \text{priority join on the network}}{C_3^{i^0} = 1}, \\
2 &: \frac{0 \& \Psi_{PrioritySync_2}(IJ_{1,2}) \& \text{priority join on the network}}{C_4^{i^0} = 2}, \quad 3 : \frac{2; \Psi_{router}(C_{1,2,3,4})}{\bar{C}_1 \wedge \neg \bar{C}_4 \Rightarrow C_4^\flat}, \\
4 &: \frac{0 \& \text{join}}{\neg \bar{C}_4}, \quad 5 : \frac{\text{coloring} \& \text{join}}{C_4^\flat \Rightarrow W_3^\flat}, \quad 6 : \frac{0 \& 3 \& 4 \& 5}{W_3^\flat}, \quad 7 : \frac{0 \& 5}{\perp}
\end{aligned}$$

This disproves the existence of the aforementioned transition meaning that when $\{BC\}$ and $\{BE\}$ are full, the request from W_1 is not ignored.

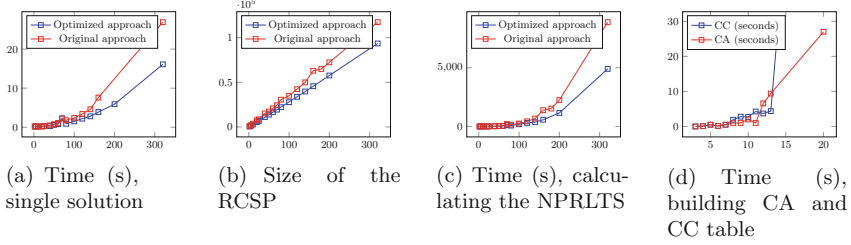
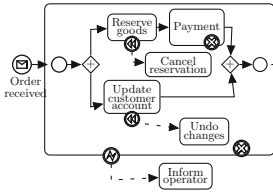
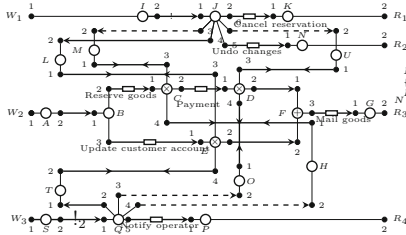


Fig. 2. Performance evaluation of N-Sequencers

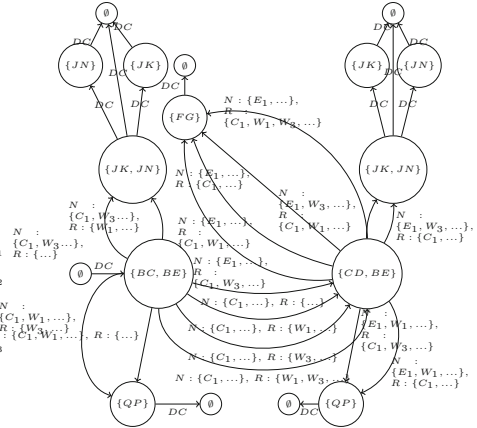
(a) A BPMN model with two priority levels



(c) Reo network of the figure 2(a)



(b) Priority agnostic NPRLTS of the figure 2(a)



The execution time of the Algorithm 1 depends on the number of states of the RCSP and the time to solve the RCSP. Thus, to study the performance of our framework and to compare it with the existing approaches, we choose *N-Sequencer*, which consists of N *FIFO* channels that are circularly connected. Adding each *FIFO*₁ channel doubles the number of states in the corresponding semantic model and increases the complexity of the constraints encoding the behavior of the network by adding new variables and new assertions on them. This makes the network a good choice for our benchmarking, where we would like to compare the solutions on state explosion. Since we are interested in comparing our approach with the existing tools, we do not include priority in our case study. This is justified by the fact that incorporating priority does not affect the number of states in the model and influences only the size of the constraint. In addition, adding more *FIFO*₁ channels to the network increases both the number of states and the size of the constraint capturing the semantics of the network. Since we use optimized third-library tools to solve the constraints, we do not distinguish

between the various form of constraints obtained from different channels and instead we observe the approximate growth of the size of constraints.

Figure 1 shows a *7-sequencer*. Though the size of the operational semantics model of this network grows in a linear fashion in relation with N , the number of intermediate states to compute the final results grows exponentially. The benchmarks have been performed on Mac Book Pro OS X El Capitan with 2.8 GHz Intel Core i7 and 16 GB MHz DDR3 memory. Our approach is implemented in Java 8. We have used Reduce Algebra System revision number 2337 to compute the conjunctive normal form of the constraints and to solve them. We have experimented with an optimization on the number of variables used in the constraints by substituting equal variables with a single variable. Figure 2(a) presents the average time to compute a single solution of the RCSP of an *N-Sequencer*. Figure 2(b) shows the relation between N and the size of the RCSP's constraints of an *N-Sequencer*. This is an indication of the complexity of the constraint. Figure 2(c) illustrates the total time required to compute all solutions of a RCSP's constraint of an *N-Sequencer*. Figure 2(d) shows the time consumed to calculate the coloring semantics and the constraint automata semantics of *N-Sequencers* using the ECT toolset. The computation of the coloring semantics and the constraint automata fail with the stack overflow error for $N = 16$ and $N = 21$, respectively. The results shows that our approach handles larger models than the existing tools can. The effect of the optimization is more significant for larger N .

6 Related Work

Several works, e.g., [10, 11, 22] use priorities to model scheduling policies. Many workflow languages rely on Petri nets [2, 4]. Priority flow in Petri net-based process models is managed with the help of inhibitor arcs and transition priorities [31]. Inhibitor arcs allow a transition to fire only if the adjacent place is empty. *Prioritized Petri nets* [9] introduce a partial order on transitions. Given a set of enabled transitions, the transitions with higher priority fire before the transitions with lower priority. Others, e.g., [28, 34] use a partial order on transitions to model priority. Our earlier approach in modeling priority using binary variables supports a limited form of priority compared to the mentioned Petri nets approaches. However, the proposed extension bridges this gap by defining priorities as non-zero natural numbers. An advantage of our model is its compositionality. Compared to the aforementioned methods, Reo fits in the realm of component-based or service-oriented architecture in a compositional way. Reo is an extensible language, where new behavioral aspects can be added. An effort to express the behavior of Reo networks via constraints is reported in [19]. It demonstrates the efficiency of the constraint-based approach. It models synchronization and data flow constraints, but no priority flow was considered. In [16], a framework is presented to encode semantics of Reo networks as CSP with predicates in the form of binary propositions and numerical constraints. An advantage of this method is handling data constraints symbolically and, hence, mitigating

the state explosion problem of automata models. We extended this framework to handle priority constraints, taking a step forward toward implementing a toolset that covers all behavioral aspects of Reo. Among the formal semantics of Reo, connector coloring comes with a limited notion of priority based on the context information. The context information affects otherwise non-deterministic data-flow choices. In [26], an automata-based semantics is proposed, which associates a preference for each transitions. A transition of lower preference is fired iff no more preferred transition can occur.

7 Conclusions and Future Work

In this paper, we addressed the problem of priority flow modelling using the Reo coordination language. We extended the unified constraint-based semantics of Reo with binary and numeric priority constraints, showed correctness of our approach for the binary case and evaluated the performance of the algorithm for solving the RCSP to derive the semantics of a Reo network given the behavior of its constituent elements. We also illustrated the use of our framework for modeling business processes with priority flow.

As part of our ongoing work, we are using this framework to encode other aspects of the semantics of Reo, specifically, timed behavior. A promising area for future work is to use our framework for constraint-based model checking of Reo networks with priority.

Acknowledgements. The publication has been prepared with the support of the “RUDN University Program 5–100” and funded by RFBR according to the research projects No. 12-34-56789 and No. 12-34-56789

References

1. Aalst, W.M.P.: Business process management demystified: a tutorial on models, systems and standards for workflow management. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 1–65. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_1
2. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: yet another workflow language. *Inf. Syst.* **30**(4), 245–275 (2005)
3. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business process management: a survey. In: van der Aalst, W.M.P., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 1–12. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44895-0_1
4. van der Aalst, W., Hofstede, A.H.M.T.: Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. Technical Report DAIMI PB-560 (2002)
5. Arbab, F.: Reo: a channel-based coordination model for component composition. *Math. Struct. in Comput. Sci.* **14**, 329–366 (2004)
6. Arbab, F.: Puff, the magic protocol. In: Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday. pp. 169–206 (2011)

7. Arbab, F., Kokash, N., Meng, S.: Towards using reo for compliance-aware business process modeling. In: ISoLA. pp. 108–123 (2008)
8. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.* **61**(2), 75–113 (2006)
9. Balbo, G.: Introduction to stochastic petri nets. In: Brinksma, E., Hermanns, H., Katoen, J.-P. (eds.) *EEF School 2000. LNCS*, vol. 2090, pp. 84–155. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44667-2_3
10. Bause, F.: Analysis of petri nets with a dynamic priority method. In: Azéma, P., Balbo, G. (eds.) *ICATPN 1997. LNCS*, vol. 1248, pp. 215–234. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63139-9_38
11. Best, E., Koutny, M.: Petri net semantics of priority systems. *Theor. Comput. Sci.* **96**(1), 175–215 (1992)
12. Bonsangue, M., Clarke, D., Silva, A.: A model of context-dependent component connectors. *Sci. Comput. Program.* **77**(6), 685–706 (2012)
13. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM (2005)
14. Butler, M., Hoare, T., Ferreira, C.: A trace semantics for long-running transactions. In: *Proceedings of the International Conference on Communicating Sequential Processes: The First 25 Years. CSP 2004* (2005)
15. Changizi, B., Kokash, N., Arbab, F.: A unified toolset for business process model formalization. In: *Proceedings of Formal Engineering Approaches to Software Components and Architectures*. ENTCS, Elsevier (2010)
16. Changizi, B., Kokash, N., Arbab, F.: A constraint-based method to compute semantics of channel-based coordination models. In: *Proceedings of the International Conference on Software Engineering Advances (ICSEA)*. IARIA (2012)
17. Changizi, B., Kokash, N., Arbab, F.: A unified toolset for business process model formalization. In: *7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010)*, pp. 147–156. ENTCS (2010)
18. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: synchronisation and context dependency. *Sci. Comput. Program.* **66**(3), 205–225 (2007)
19. Clarke, D., Proenca, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. *Sci. Comput. Program.* **76**(8), 681–710 (2011)
20. Dijkman, R., Hofstetter, J., Koehler, J. (eds.): *BPMN 2011. LNBIP*, vol. 95. Springer, Heidelberg (2011). <https://doi.org/10.1007/978-3-642-25160-3>
21. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: *Fundamentals of Business Process Management*. Springer, Berlin (2013). <https://doi.org/10.1007/978-3-662-56509-4>
22. Füricht, R., Prähofer, H., Hofinger, T., Altmann, J.: A component-based application framework for manufacturing execution systems in *c#* and *.net*. In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, pp. 169–178. CRPIT 2002, Australian Computer Society, Inc. (2002)
23. Havey, M.: *Essential Business Process Modeling*. O'Reilly Media Inc., Newton (2005)
24. Jongmans, S.-S.T.Q., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Automatic code generation for the orchestration of web services with Reo. In: De Paoli, F., Pimentel, E., Zavattaro, G. (eds.) *ESOCC 2012. LNCS*, vol. 7592, pp. 1–16. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33427-6_1

25. Jongmans, S., Arbab, F.: Overview of thirty semantic formalisms for Reo. *Sci. Ann. Comput. Sci.* **22**, 201–251 (2012)
26. Kappé, T., Arbab, F., Talcott, C.L.: A compositional framework for preference-aware agents. In: Proceedings of the The First Workshop on Verification and Validation of Cyber-Physical Systems, V2CPS@IFM 2016, Reykjavík, Iceland, 4–5 June 2016, pp. 21–35 (2016)
27. Kokash, N., Arbab, F.: Formal design and verification of long-running transactions with extensible coordination tools. *IEEE Trans. Serv. Comput.* **6**(2), 186–200 (2013)
28. Lomazova, I.A., Popova-Zeugmann, L.: Controlling petri net behavior using priorities for transitions. *Fundam. Inform.* **143**(1–2), 101–112 (2016)
29. Lu, R., Sadiq, S.: A survey of comparative business process modeling approaches. In: Abramowicz, W. (ed.) *BIS 2007*. LNCS, vol. 4439, pp. 82–94. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72035-5_7
30. Meng, S., Arbab, F.: Web services choreography and orchestration in Reo and constraint automata. In: Proceedings of the ACM Symposium on Applied Computing, pp. 346–353. ACM Press (2007)
31. Padberg, J.: Reconfigurable petri nets with transition priorities and inhibitor arcs. In: Parisi-Presicce, F., Westfechtel, B. (eds.) *ICGT 2015*. LNCS, vol. 9151, pp. 104–120. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_7
32. Reisig, W.: *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer, Berlin (2013). <https://doi.org/10.1007/978-3-642-33278-4>
33. Schumm, D., Turetken, O., Kokash, N., Elgammal, A., Leymann, F., van den Heuvel, W.-J.: Business process compliance through reusable units of compliant processes. In: Daniel, F., Facca, F.M. (eds.) *ICWE 2010*. LNCS, vol. 6385, pp. 325–337. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16985-4_29
34. Valero, V., MaciÀ, H., Pardo, J.J., Cambronero, M.E., DÁaz, G.: Transforming web services choreographies with priorities and time constraints into prioritized-time colored petri nets. *Sci. Comput. Program.* **77**(3), 290–313 (2012). <http://www.sciencedirect.com/science/article/pii/S0167642311001407>, feature-Oriented Software Development (FOSD 2009)