Constraint-based analysis of business process models

Changizi, B.

Author: Changizi, B.
Title: Constraint-based analysis of business process models
Issue Date: 2020-02-25

Constraint-Based Analysis of Business Process Models

Proefschrift

te verkrijging van
de graad van Doctor aan Universiteit Leiden,
op gezag van Rector Magnificus Prof. Mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen 25 February 2020
klokke 13:45

door

Behnaz Changizi

Geboren te Hamedan, Iran, in 1979

Promotor: Prof. Dr. F. Arbab

Copromotor: Dr. N. Kokash (Peoples' Friendship University of Russia)

Promotiecommissie:

Prof. Dr. F.S. de Boer
Prof. Dr. A. Plaat
Prof. Dr. M. Sirjani (Malardalen University)
Prof. Dr. A. Lazovik (University of Groningen)
Dr. M.M. Bonsangue

The work in this thesis has been carried out at Centrum Wiskunde & Informatica and Leiden University, and under the auspices of the research school IPA: Institute for Programming research and Algorithmics. The research was partially funded by the EU project EU FP7 IST project COMPAS: Compliance-driven Models, Languages, and Architectures for Services.

Contents

Listings

List of Figures

List of Tables

1

Introduction

The term *business process modeling* is first introduced by S. Williams [Wil67] where he argues that the techniques for modeling physical control systems could be applied to business processes [DM03]. However, it took until the 1990s for the term *business process* to become popular [Hoo11].

At the time, companies started to think in terms of processes rather than functions and procedures [Rol95]. Process thinking ensures the right development direction by analyzing the chain of events in an organization. Examples include the events occurring from purchase to supply or from receiving orders to sales.

A *business process* is a set of related and structured activities, which serves a specific goal for a customer [Rol95]. The de-facto standard in the field of business process modeling is the Business Process Model and Notation (BPMN).

Business Process Model and Notation (BPMN) [Gro11], previously referred to as Business Process Modeling Notation, is a graphical representation of business process models based on flowcharting techniques. The main goal of BPMN is to provide an understandable notation for both technical experts and business users.

Similar to many modeling languages, it is possible for a BPMN model to contain errors. Syntactical errors are created by connecting the modeling elements in an invalid manner. In general, syntactical errors can be detected simply by parsing the model [AP08]. A number of BPMN designing tools such as Eclipse BPMN Modeler [BPM], ARIS Express [ari], and Yaoqiang [Yao] can detect syntactical errors in models.

However, a model may contain behavioral errors, which are more complicated to detect. For instance, a model may represent a process that is not *sound*. A process is sound when every reachable state from an initial state has a way to reach a final state [GPR+07]. A process may contain *deadlock* or *livelock*. Deadlocks occur when a process can reach a non-final state that it cannot leave. Livelocks happen when a process ends in one path, but some states are still active with no progress possible. Detecting behavioral errors requires investigating the runtime behavior of a process. An informal approach to finding cases of deadlock and livelock in BPMN models has been proposed in [AP08] [TJ10], which is based on finding the pre-defined patterns of such errors in the model. Although this approach has low computational costs, it is not complete in term of finding other forms of errors.

Formalizing semantics of a BPMN process enables automated model checking of the process in order to detect behavioral errors. Similar to a variety of BPM systems on the market [DRMR13], the foundation of BPMN is based on Petri nets [vdA04]. The choice of Petri nets as foundation for BPM system implementation over other formal methods, often more expressive or specialized [RBM05, BHF05], is not surprising: hardly any model is as simple, intuitive, and naturally supports task traceability.

While undoubtedly Petri nets based models enable automated process analysis within BPM systems, they lack few desirable characteristics: i) They lack compositionality, which means that they cannot deal with large and complex systems. Ideally, we would like to plug semantic models for individual components 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 operational semantics of the model and generate incompatible dialects of process-specification languages adopted by various tools.

An alternative theory for coordinating concurrent components is called the Reo coordination language [Arb04]. Reo has been used to formalize semantics of Business Process Modeling Notation (BPMN) [AKM08b], UML Activity and Sequence Diagrams [CKA10], map BPEL fragments [STK+10], represent transactional workflows [KA13], implement service orchestrations [JSS+12] and service choreographies [MA07b].

In this dissertation, we propose formal semantics for Business Process Modeling Notation (BPMN) models in terms of Reo. The mapping of BPMN to Reo is implemented as a plugin in the Reo analysis tool-set in a model-driven paradigm. Our mapping completes the proposed mapping of BPMN to Reo in [AKM08b] by covering not only basic BPMN constructs, but also advanced structures such as BPMN transactions. In addition, our proposed mapping rules are expressed formally in a dedicated language for model to model transformation.

Since synchronization propagates through composition in Reo, it allows composition of components and services in an intuitive way, and addresses the issue (i) mentioned above. Reo is easily extensible to support more advanced process models, such as timed [MA07a] or stochastic workflows networks [MSKA10], via defining new channels. However, the open-ended nature of Reo channels makes 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 [JA12]. They vary from rather simple that cover basic Reo behavior (e.g., constraint automata [BSAR06]) to more complex models that capture specific behavioral aspects, e.g., context-sensitivity [CCA07]. In some of these semantic models, computing the overall semantics of a system given automata-based semantics for its parts (components, services or glue code) is computationally expensive. This hampers using the language for analyzing large real-world business processes.

In this dissertation, we present a constraint-based framework, which unifies various formal semantics of Reo. In this framework, the behavior of a Reo network is described as a constraint satisfaction problem (CSP). A CSP is a problem whose solutions must satisfy some limitations also known as constraints. The constraint-based nature of our approach allows simultaneous coexistence of several semantics in a simple fashion. The behavior of a Reo network is determined by the solutions to its CSP. Since any solution must satisfy all the encoded formal semantics, the framework eliminates any behavior inconsistent with (an aspect of) a formal semantics of Reo.

Another advantage of our proposed constraint-based approach compared to the existing approaches of deriving formal semantics of Reo is its efficiency due to efficient constraint solving methods and optimization techniques used in the off-the-shelf constraint solvers. We support this claim with a case study.

Among the behavioral aspects required to model a business process is *priority*. The notion of priority is necessary for modeling behaviors such as transaction and exception handling, where the data flow representing the error or exception should interrupt the normal flow. A formal semantics to model priority in Reo, named

Constraint Automata with Priority (CAP), has been proposed in [ABS15]. CAP provides means to model propagation and stopping the propagation of priority. Despite its comprehensive approach in modeling priority, the proposed semantics is computationally expensive for direct implementation.

Inspired by CAP, in this dissertation, we present an alternative approach to model priority in Reo by extending our constraint-based framework with priority-aware premises. Further, we extend our priority-aware formal model to support not only a binary notion of priority, as modeled in CAP, but also numeric priorities.

1.1 Contributions

The contributions of this dissertation are as follows:

- We present a model-driven mapping of business process models specified in BPMN into Reo networks. Such transformations enable application of automated analysis and model checking on business processes. We have implemented our proposed mapping in a rule-based fashion using a dedicated transformation language, which makes the implementation of the mapping concise, readable, and easy to maintain. We have integrated our mapping into the Extensible Coordination Tool-set (ECT), the integrated development environment for Reo. This makes it easier for business process models to be fed to various tools in ECT. Figure 1.1.1 shows an example of a BPMN model with the option to be converted to Reo using our BPMN to Reo plugin. Figure 1.1.2 depicts the generated Reo network.

- We provide an extensible constraint-based approach to unify various semantic models of Reo networks. We represent a problem of computing semantics for a complex Reo network by encoding semantics of individual channels as constraints and solving the corresponding constraint satisfaction problem. This approach bridges the expressiveness gaps and incompatibility among different Reo semantics. In addition, using a constraint-based approach replaces direct implementations of algorithms for calculating different Reo formal semantics.

- We extend our constraint-based framework to support the priority-aware behavior of Reo connectors. Priority is an important concept in modeling transactions. Our work makes it more straight-forward and less complicated to obtain Constraint Automata with Priority (CAP) formal semantics for Reo. Our framework is the only existing approach that integrates various behav-

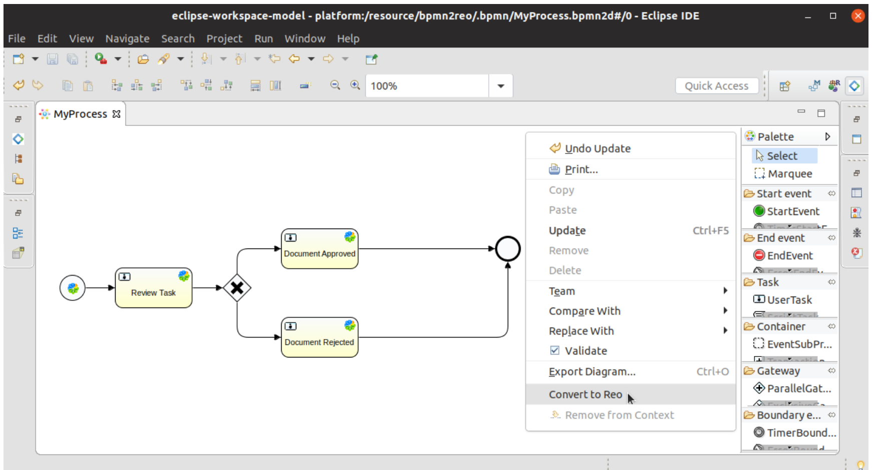


Figure 1.1.1: The BPMN to Reo converter menu in ECT

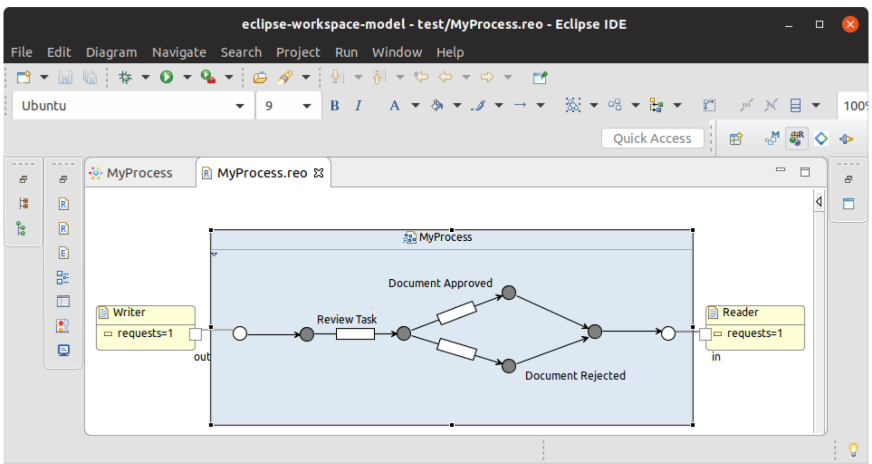


Figure 1.1.2: The mapping of Figure 1.1.1

ioral aspects of a Reo network (e.g. data-dependency, context-sensitivity, priority-awareness) under one umbrella.

1.2 Outline

The rest of this dissertation is organized as follows:

- In Chapter 2, we introduce BPMN 2 modeling elements and introduce an example of BPMN with problems. This chapter is based on the technical content of [1], listed in Section 1.3.

- Chapter 3 provides an overview of the Reo coordination language. There, we describe the behavior of Reo primitives in an informal style.

- Chapter 4 contains an overview of several formal semantic models proposed for describing behavior of a Reo connector. The definitions of the semantics that are relevant to this work are given in details.

- Chapter 5 describes our rule-based model-driven approach in transforming BPMN models to Reo connectors. The transformation handles advanced BPMN elements, namely, transaction and compensation.

  An obstacle in computing execution semantics of some BPMN models with high-level elements such as transaction is that their behavior is too complicated and elaborated to directly be mapped to constructions of a language used for verification. To tackle this issue, we suggest a refinement procedure to substitute such high-level constructs with a set of simpler elements that together deliver the same functionality. This chapter is partially based on the technical content of [1], listed in Section 1.3.

- In Chapter 6, we introduce our constraint-based framework to capture the formal semantics of Reo networks, given by two different formal semantics namely, Constraint Automata with State Memory (CASM) and Connector Coloring (CC) [CCA07]. CASM is an extension of Constraint Automata (CA), which is one of the most popular semantics for Reo. We favor using CASM over CA, which is a simpler semantics, because CASM provides a mechanism to model the state values. This helps in treating the states symbolically. Therefore, unlike CA every data-item entering a buffer does not lead to a new state.

  To capture context sensitivity, a behavioral aspect that CA and some of its extensions miss, we use CC, which models context sensitivity in a Reo connector using graph coloring techniques.

  We present a tool to generate CASMs from Reo networks in a compositional manner, where the part of behavior that is not compliant with CC is ruled out.

  We employ highly optimized off-the-shelf constraint solvers instead of straightforward custom algorithms for computing the semantics [CKA12]. We provide formal arguments to show the correctness of our approach. Then, we present an evaluation on the performance of our framework through a case study. The technical content of [4], listed in Section 1.3 is the basis of this chapter.

- In Chapter 7, we extend our framework to support priority and its propagation through a Reo connector. We propose a constraint-based solution to replace the custom algorithm to calculate the priority-aware behavior of a Reo connector [CKA19]. We first introduce a binary model of priority and show how it can be encoded in our constraint-based framework. Subsequently, we extend this solution to numeric priorities. We show the application of our model in a case study. This chapter is based on the technical content of [5], listed in Section 1.3.

- Chapter 8 concludes this thesis and outlines future research directions.

1.3 Publications

1. Behnaz Changizi and Natallia Kokash and Farhad Arbab. A Unified Toolset for Business Process Model Formalization. 7th International Workshop on Formal Engineering approaches to Software Components and Architectures, pages 147-156. ENTCS, 2010.

2. Behnaz Changizi and Natallia Kokash and Farhad Arbab. A Semantic Model for Service Composition with Coordination Time Delays. International Conference on Formal Engineering Methods, pages 106-121. 2010.

3. Behnaz Changizi and Natallia Kokash and Farhad Arbab. Input-Output Conformance Testing for Channel-based Service Connectors. In: Proceedings of PACO, pages 19–35. 2011.

4. Behnaz Changizi and Natallia Kokash and Farhad Arbab. A Constraint-based Method to Compute Semantics of Channel-based Coordination Models. International Conference on Software Engineering Advances. IARA, 2012.

5. Behnaz Changizi and Natallia Kokash and Farhad Arbab. Service Orchestration with Priority Constraints. International Conference on Fundamentals of Software Engineering, pages 194-209. LNCS, 2019.

2

Business Process Model and Notation

2.1 Introduction

Business Process Model and Notation (BPMN) [Gro11], also known as Business Process Modeling Notation, is a standard graphical representation of business process models. BPMN bridges the gap between visualization of the business processes and their actual implementation by providing an understandable notation for both business stakeholders and technical experts.

BPMN is based on flowcharting techniques. It allows modeling complex business processes using its diverse set of control structures, which covers concepts such as sequencing, repetition, choice, concurrency, messaging, failure, transactions, etc. BPMN has an expressive notion to define events and to associate triggers to the defined events. Furthermore, it provides means to form reusable units out of a set of elements.

The first version of BPMN is developed by the Business Process Management Initiative (BPMI) in 2004. In 2005, BPMI and the Object Management Group (OMG) merged. BPMN is maintained by OMG since then. In 2006, the BPMN specification was adopted as an OMG standard. In 2011, the final edition of BPMN

2 specification was released.

BPMN 1.2 presents a notation for modeling business processes and informally expresses the semantics of the modeling primitives. This leads to ambiguity and confusions in interpretation of a process. For instance, the authors in [vdADK02] present a deadlock situation called *vicious circle* that is caused by using convergent *inclusive gateway*s. This is a class of situations where two *inclusive gateway*s are connected in a cyclical way. Moreover, BPMN 1.2 specification provides no details on model serialization format.

BPMN 2, the biggest revision of BPMN so far, presents a formal definition in terms of a meta-model, that is a formal definition of the constructs and their relations in a valid model. The meta-model specifies a serialization format that enables model exchange among different BPMN 2 tools. In the context of modeling elements, BPMN 2 offers the following enhancements over previous versions:

- It expands the set of BPMN gateways with *exclusive* and *inclusive* event-based gateways.

- It enriches the set of activities by adding *business rule task*, *sequential multi-instance* activity, *event sub-process* that handles events occurring in bounding *sub-process*, and *call activity* that invokes a global *sub-process*.

- It enhances *event*s by introducing *escalation*, and *complex events*, and the concept of *interrupting* and *non-interrupting* events.

Although BPMN 2 provides an explicit execution semantic, the semantics are expressed in informal fashion. This leaves rooms for interpretation for a number of issues such as deadlocks and race conditions.

In this chapter, we provide an overview to BPMN. We also present examples of process models containing semantical errors.

2.2 BPMN 2 elements

A BPMN diagram consists of a number of elements that fall into the categories of *flow objects*, *connecting object*s, *swimlane*s, and *artifacts*. A flow object can be an *event*, a *gateway*, or an *activity*.

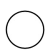
2.2.1 Connecting objects

Connecting objects are used to connect the other BPMN elements:

- *Sequence flow*s represent the occurring order of processes in a business model.
- *Message flow*s are used to exchange messages between process participants.
- *Association flow*s associate modeling elements to each other. For instance, a *compensation task* is associated to its task via an association flow.

2.2.2 Events

Events represent triggers occurring during execution of business processes. Events usually have a *cause* or a *result*. The representation of an *event* is a circle wherein internal markers are placed to denote triggers or results. Based on the time that events affect the flow, they fall into three categories:

 *Start events*, which start a process;

 *Intermediate events*, which occur between start and end of a process;

 *End events*, which terminate a process.

Each time a process receives a new start event trigger, a new instance of the process begins to execute. Therefore, a process may have many process instances. Start events and intermediate events are *catching*, meaning that they catch a trigger in order to occur. End events and some of intermediate events are *throwing* as they throw a result. Compared to the passive nature of catching events, throwing events are *active* as they trigger themselves rather than waiting for a trigger to take place.

The following intermediate events can attach to the boundary of an activity: *message*, *timer*, *error*, *compensation*, and *signal*. In this case, they can only occur while the surrounding activity is active. *Boundary event*s can either be interrupting or non-interrupting.

Interrupting events stop the execution of the activity and direct the flow out of the boundary event, while non-interrupting events do not interfere with the execution of the activity. Instead, they start the flow out of the boundary event in parallel. Another difference is that non-interrupting events can occur several times while the surrounding activity is running.

Following is the list of event types in BPMN 2:

A *none* event has no defined trigger. It can indicate a start point, a state change or a final state. Each process can only have one *none start event*.

A *message* event is used to model exchange of messages. A message has a specific receiver.

A *signal* is broadcasted between processes. It differs from message in that a message has a specific target, but a signal is broad-casted. A thrown signal can be caught multiple times.

A *timer* event indicates a waiting time within the process. A timer trigger can be a specific date/time value or a duration.

A *conditional* event occurs when a business condition becomes true.

A *link* is a mechanism for connecting two sections of a process. A throwing link event is used at the exit point, while a catching link event as the entrance point. Using link helps keeping the model clean and prevents spaghetti models.

A *cancel* event is always used with a transaction sub-process. It indicates that the transaction should be canceled. A cancel event triggers a cancel intermediate event attached to the sub process boundary.

A *terminate* event indicates that all activities in the process should be immediately ended. In this case, the process is ended without compensation or event handling.

A *throwing compensation* event indicates that a compensation is needed. A catching compensation event states that a compensation will occur when the event is triggered. All other boundary events occur only while the activity that they are attached to is active. In contrary, an attached compensation takes place only if the process triggers a compensation and if the activity to which compensation is attached has been completed successfully.

A *multiple* event summarizes several events with a single event. A catching multiple event occurs if at least one of its specified events occurs. However, a throwing multiple triggers all the defined events.

A *parallel multiple* event, which is added in BPMN 2, is a supplement to multiple event. A parallel multiple event is only catching. It indicates that all of the defined events are required in order to trigger this event.

Escalation is new in the BPMN 2 specification. An escalation event is used to trigger a path in middle of a process flow that requires involvement of a higher responsibility.

Based on the types, event triggers are forwarded in five different strategies:

- *Publication*: A published trigger can be caught by any catching event that matches the trigger within any scope where it is published. *Message* and *signal* events triggers are forwarded this way.

  Messages are created out of the pool wherein they are published. In case that a message should be received by a specific process instance, the particular instance in referred by the message.

  Signals are created inside the pool wherein they are published. In general, signals are used to broadcast within and across processes, pools, and process diagrams.

- *Direct Resolution*: The timer and conditional triggers are thrown implicitly. These triggers wait for a defined time or a specific condition to trigger the related catch event, respectively.

- *Propagation*: A propagated trigger is forwarded from its origin to the innermost enclosing level that has an attached catching event that matches the trigger. Instances of events that propagate are *error* and *escalation*.

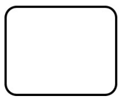
  Unlike error triggers that are critical and suspend execution, escalations are non-critical and allow execution to proceed normally. If there is no catching event found for an error or an escalation trigger, the trigger is unresolved.

- *Cancellation*: When a *cancellation* occurs, all running activities terminate and all activities in the sub-process wherein cancellation applies are compensated, if they are completed successfully. In case that the sub-process is a transaction, it needs to be rolled back.

- *Compensation*: A successfully completed activity is *compensated* by its compensation handler, which is either user-defined or implicit. In latter case, the compensation handlers of the enclosed activities are invoked in the reverse order of their execution. If an activity has not completed successfully, nothing happens and no error is raised.

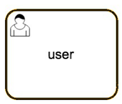
2.2.3 Activities

An activity describes the type of work that needs to be done. An activity is either a task, a sub-process, or a transaction. BPMN represents the activity in a high-level of abstraction. It is not the BPMN responsibility to describe the activity details.



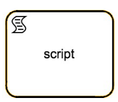
Tasks, which are atomic activities have several types:

A *manual task* is a task that is performed manually.

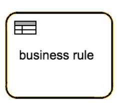


A *user task* is performed by a person with assistance of automation.

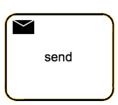
Service tasks are services such as web services or automated applications.



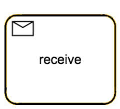
A *script task* is executed by a business process engine.



Business rule tasks are introduced in BPMN 2. They are performed by business rule engines.

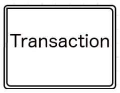


A *send task* is a simple task with an outgoing message flow, which is used for sending messages. The task is completed after the message is sent.

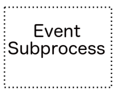


A *receive task* is a simple task with an incoming message flow, which waits for a message to arrive. Once it receives the message, the task is completed.

A *sub-process* captures a set of activities, gateways, and flows within a single activity. It hides or reveals details of business process based on being expanded or collapsed, which is denoted using a plus sign at the bottom of the sub-process. A sub-process may only begin with a *none start* event and end with a *none end* event.



A *transaction* is a sub-process that all of its enclosed activities constitute a logical unit of operation, meaning that all the activities must be completed successfully, and if one fails, all of them need to be compensated.



Event sub-process are introduced in BPMN 2. An event sub-process behaves like a boundary event, but it resides inside a process or a sub-process rather than on their boundaries.
An event sub-process can be considered as an optional sub-process that occurs when its start event is triggered.
Similar to boundary events, an event sub-process may interrupt the containing process or run in parallel in a non-interrupting fashion, depending on the type of its start event.
In addition, it is allowed to have only one start event that is non-empty. The event types that can be used as a start event for an event sub-process are: *message*, *conditional*, *signal*, *timer*, *escalation*, *error*, *multiple*, and *parallel-multiple*. As mentioned, the only way to run an event sub-process is by triggering its start event. As a result, no incoming or outgoing sequence flow can connect to an event sub-process.

In BPMN 1.2, there are two types of sub-processes: *embedded* and *reusable*. BPMN 2 sub-processes are inherently embedded. They can only be reused if they are defined globally and are referenced by call activities.

An embedded sub-process can only contain a *none start* event. It cannot have other types of start events such as timers or messages.

Furthermore, an embedded sub-process can only be found inside a process to which it belongs. A global sub-process, on the other hand, can reside within different processes.

In BPMN 2, reusable task and sub-processes are invoked using a *call activity*. According to the BPMN 2 specification [Gro11], a call activity in BPMN 2 corresponds to the BPMN 1.2 reusable sub-process, while a sub-process in BPMN 2 corresponds to the BPMN 1.2 embedded sub-process.

A transaction has three possible outcomes:

- All the activities finish *successfully*. In this case, the process proceeds with the normal flow.

- In case of a *failure*, the compensation tasks associated to the successfully completed activities execute. The process continues through the cancel intermediate event.

- In case that an *unexpected error* takes place, the sub-process activities are interrupted without any compensation. The process then proceeds with the intermediate error event.

An activity can be annotated using different *markers* that indicate the nature of the activity. The markers are as follows:

- The *loop* marker indicates that the attached activity executes multiple times until the loop condition holds. The condition can be evaluated either in the beginning or in the end of the activity depending on a specific attribute of the activity.

- A *compensation* marker is used to undo a completed activity.

- A *sequential multi-instance* marker defines an activity that has multiple instances created sequentially. The number of instances to be instantiated is either defined as an attribute of the activity or as the cardinality of input data items.

- A *parallel multi-instance marker* represent activities that can be executed in parallel as multiple instances. Each instance can have a different set of input parameters.

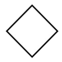
- An *ad-hoc marker* is used to represent an activity, whose inner tasks have no required order. Each task can start at any time. There is no dependency among the activities.

2.2.4 Gateways

Gateways manage the control flows within a process or sub-process by specifying the interaction among sequence flows as they converge and diverge. The list of BPMN 2 gateways follows:

2.2.5 Swimlanes and artifacts

A *swimlane* is used for organizing and categorizing activities inside a business process. A *swimlane* can be either a *pool* or a *lane*. A *pool* represents a participant in a business. *Lane*s are partitions inside a *pool*.



*Data-based exclusive gateway*s are used to create alternative paths based on the conditions that are set on the incoming data flow. A diverging exclusive gateway, also called *decision*, routes the incoming flows to one of the mutually exclusive alternative outgoing flows. A converging exclusive gateway directs one of its incoming flows to its only outgoing flow.

*Data-based inclusive gateway*s create alternative but also parallel paths within a process flow. A diverging inclusive gateway directs its incoming flow to one or more outgoing flows based on conditions. A converging inclusive gateway, on the other hand, awaits incoming flows to complete.

*Parallel gateway*s are used to create and also to combine parallel flows. A diverging parallel gateway creates parallel flows, while a converging one merges the incoming flows into one outgoing flow.

Event-based gateway routes based on occurrence of events rather than on data. In addition to events, it also works with receive message task. An event-based gateway is always followed by catching events or receive tasks.

A *parallel event-based Gateway* is similar to a parallel data-based gateway with the difference that it depends on occurrence of events rather than on data.

A *complex gateway* models complex synchronization behavior. An expression is used to describe the behavior of the gateway.

*Artifact*s are used for adding information into the model. The followings are three types of *artifact*s:

- *Data object*s, which describe the required or the produced data in an activity.

- *Group*s are used to categorize different activities.

- *Annotation*s are providing information about the model.

Example 2.2.1 *Figure 2.2.1 depicts a BPMN model consisting of two processes. The* receiver *process starts, waits till receiving a message from the* sender *process before it ends. While the* sender *process starts, evaluates a condition based on which it chooses to end or to send a message to the* receiver *process, and returns back to the condition evaluation step.*

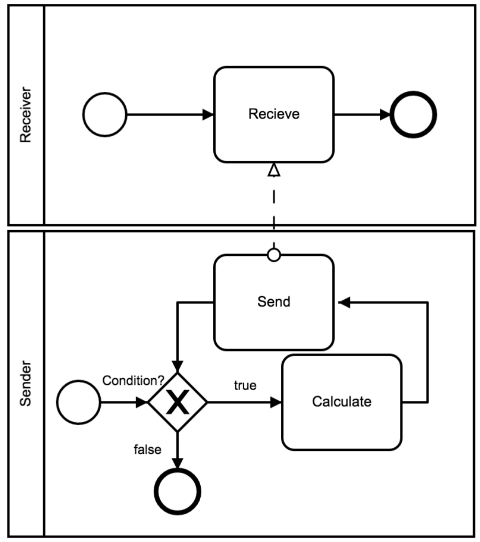


Figure 2.2.1: An example of messaging in BPMN

The desired behavior of this model is that the processes start, the message exchange occurs, and they end. However, it is possible that the sender *process finishes without sending any message. In this case, the* receiver *process keeps waiting for a message that will never arrive. This is an example of* deadlock.

In addition, the model contains a livelock, *which occurs if after the* receiver *process receives a message from the* sender *process and finishes, the* sender *keeps going back to the sending step and does not end.*

3

Reo Coordination Language

3.1 Introduction

In the realm of service-oriented programming that is a current trend in software development, the behavior of a software system is not only defined by the functionalities of its underlying services, but also in terms of their interactions. The code written to realize the latter is often referred to as *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 increase over time. This makes these systems hard to modify and maintain. Coordination languages offer a more manageable alternative for generating glue code.

Reo [Arb04] 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 in terms of synchronization, buffering, mutual exclusion, priority, etc.

The primitive constructs of Reo are *channel*s. Each *channel* has two *end*s, also called *port*s. Channel ends are either of type *source* that read data into the channel or *sink* that write the channel's data out.

Channels can connect to each other on their ends to form compound elements. Reo *connector*s, also called *network*s are constructed this way. A Reo *node* is formed by one or more channel ends.

Furthermore, Reo provides a mechanism for hierarchical modeling and abstracting from inner structures by means of *component*s [Arb04]. A connector can turn into a *component*. In this case it will exhibit (part of) its inner logic as an observable behavioral interface.

Reo emphasizes on the connectors and their compositions rather than the entities that connect to the connectors to coordinate with each others. A Reo connector imposes a specific coordination pattern on interactions occurring between entities. This happens without the entities controlling or being necessarily aware of this pattern. This type of coordination is called *exogenous*, as it is performed from the outside.

According to a survey of coordination languages [Arb06], Reo belongs to the class of dataflow-oriented coordination languages, which is between the data-oriented and the control-oriented classes.

While the main concern of data-oriented coordination languages is consistency among shared data, control-driven languages focus on the flow of control. In comparison, dataflow-oriented languages define the communicating entities, the points of data-flow, and exchanging data-items.

3.2 Reo

In this section, we present an informal overview of the pre-defined set of Reo constructs. Following is the list of Reo channels:

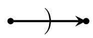
A *sync* channel has a source and a sink end. It accepts data from its source end iff it can dispense it simultaneously through its sink end.

A *lossySync* has a source and a sink end. It reads a data-item from its source end and writes it simultaneously to its sink end. If the sink end is not ready to accept the data-item, the channel loses it.

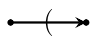
A *syncDrain* has two source ends and no sink end. It reads data through its two source ends iff both ends are ready to interact simultaneously. The channel discards the received data-items.

A *syncSpout* has two sink ends and no source end. For each sink end, the channel generates a data-item out of the underlying data domain and writes them simultaneously to the corresponding ends.

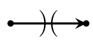
An *asyncDrain* has two source ends and no sink end. It accepts and discards a data-item from either of its source ends that offers data. If both ends offer data-items simultaneously, the channel chooses one of the ends non-deterministically.



A *blockSourceSync* channel is a *Sync* channel that blocks the propagation of priority from its source end toward the sink end.
This channel and the two next priority blocking channels are used to limit the scope affected by priority, which originates from a *PrioritySync* channel.

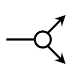


A *blockSinkSync* channel is a *Sync* channel that stop spreading of priority from its sink end toward the source end.

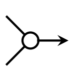


A *blockSync* channel is a combination of *BlockSourceSync* and *BlockSinkSync*. It stops the propagation of priority in both directions.

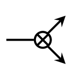
The following is a list of pre-defined Reo components that are abstracted connectors.



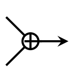
A *replicator* has one source end and one or more sink ends. It replicates data-items coming from its source to its sink ends simultaneously.



A *merger* has one or more source ends and a sink end. It chooses one of its source ends that is ready to communicate in a non-deterministic way, receives the incoming data-item, and writes it to its sink end simultaneously.



A *router* has one source end and one or more sink ends. It accepts a data-item from its source end and simultaneously replicates it on one of its sink end that is non-deterministically chosen from its set of sink ends, which are ready to accept data.



A *cross-product* has one or more source ends and a sink end. It accepts a data-item from each of its source ends. Furthermore, it forms a tuple of the data-items that are set in the counter-clockwise order with respect to the sink node. It writes the tuple on its sink end. All of these operations occur simultaneously.

As mentioned, Reo nodes are created from channel ends. In case that the node only consists of source ends, it is called a *source* node. A node is *sink*, if it is formed by merely sink ends. Otherwise, if a mixture of source and sink ends collide, the created node is called a *mixed* node.

A mixed node is an atomic combination of a replicator and a non-deterministic merger. Each read and write action needs all of its involved source and sink ends to be able to interact synchronously. Otherwise, the action cannot take place.

3.3 Examples

Example 3.3.1 *Figure 3.3.1 shows a Reo network that is composed of a* lossySync *and a* FIFO_1 *channel. When the* FIFO_1 *channel is empty, the* lossySync *reads a value from its source end and passes it to its sink end that coincides with the source end of the* FIFO_1 *channel. Therefore, the* FIFO_1 *channel becomes full. The data stored in the* FIFO_1 *channel can be read and consumed via its sink channel. Before that the* FIFO_1 *channel loses its data, the* lossySync *channel accepts but loses all its incoming data.*

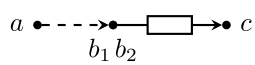


Figure 3.3.1: An example of a context-dependent Reo network

Example 3.3.2 *Figure 3.3.2 depicts a Reo network consisting of two* filter *channels with negating conditions. The first channel reads a data item from its source end and writes it on its sink end if it matches its condition, otherwise it loses the data. In the former case, the data item will not satisfy the condition corresponding to the second channel, so it is lost by the second channel. In both cases, there won't be any write operation on the sink end of the second channel.*

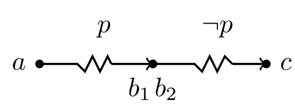


Figure 3.3.2: An example of a data-aware Reo network

Example 3.3.3 *Figure 3.3.3 illustrates a Reo network containing two* FIFO_1 *channels. The network behaves as a* FIFO_2 *buffer. In the beginning, both channels are empty. If there is an incoming data item on the source end of the first channel, the* *channel accepts the data and becomes full. Then, by an internal transition the data item is moved to the second channel. It makes it possible for the first channel to read another data item and/or to writes out the stored data through the sink end of the second channel.*

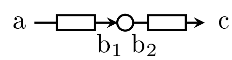


Figure 3.3.3: A Reo network for a $FIFO_2$ buffer

3.4 Extensible Coordination Tools (ECT)

A variety of Reo related tools are bundled together in a common framework, called Extensible Coordination Tools (ECT) [AKM+08a]. The tools in the framework are integrated as Eclipse plugins and operate based on the operational semantics of Reo, most notably, connector coloring and variations of constraint automata. ECT includes tools to design, transform, animate, model check, test, perform QoS analysis, and generate executable code from Reo connectors.

The ECT tools can be chained together to enable analysis on business process models. Here, we briefly overview these tools:

- *Graphical editor*: The graphical editor provides facilities to design Reo networks. The editor has been implemented based on the Eclipse Modeling Framework (EMF) [SBPM09] and Eclipse Graphical Modeling Framework (GMF). As a requirement of the model-driven approach and to work with EMF, Reo meta-model has been defined in [Kra11] [KMLA11].

- *Animation tool*: The animation tool produces simulation of Reo networks in the format of Adobe Flash [fla]. The tool is based on the animation semantics introduced in [Cos10] and visualizes the token game in Reo connectors [Kra11].

- *Verification tool*: Vereofy [BBK+10] is a model checker for Reo networks developed at the Technical University of Dresden. It can be used independently or from the ECT.

- *mCRL2 conversion tool*: Another model checker for Reo networks that is integrated into ECT is the mCRL2 [GMR+06]. The mCRL2 to Reo converter tool translates constraint automata specifications of Reo into mCRL2 specifications.

- *Execution engines*: ECT includes two execution engines: i) The centralized execution engine of Reo is a code generation framework based on constrained automata [BSAR06]. ii) The distributed execution engine for Reo is implemented based on constraint-based semantics of Reo [Pro11].

- *The Extensible Automata (EA) framework*: Extensible Automata (EA) framework is a unified framework for generating automata-based semantics of Reo networks. The framework comes with a graphical automata editor, which also can be used outside of the context of Reo. It includes functionality to generate automata models with stochastic information from graphical Reo models. From these models, it is possible to extract Continuous Time Markov Chains (CTMCs) that can be analyzed by the external tools such as PRISM probabilistic model checker [KNP02] or ECT stochastic simulation tool [Kan10].

- *BPMN 2 to Reo conversion tool*: In the context of this thesis, we have implemented a plugin to convert BPMN 2 models into Reo connectors [CKA10]. The converter deals with transactions, whose behavior is relatively more complex to map, in a two phases manner.

  The first phase is refinement, wherein transactions are substituted by a group of BPMN 2 elements, which collectively presents the same behavior as the transaction, yet they are easier to be mapped to Reo. In the second phase, the BPMN 2 constructs are being matched against some patterns to generate corresponding Reo elements. Chapter 5 elaborates on the converter.

- *Constraint-based semantics calculator*: As part of this thesis, we have implemented a tool to generate data-dependent, context-sensitive, and priority-aware formal semantics of Reo. To generate the automata-based formal semantics of Reo networks, we express the behavior of the Reo network in term of constraint satisfaction problem. From the solutions to this problem, we build the automata model.

  Our approach in using constraint solving to get the semantics of a Reo network is similar to the one used to generate the distributed execution engine for Reo [CPLA10]. However, unlike [CPLA10] [Pro11] , we support data, time, and priority. Another difference is that we calculate the all the possible behavior, while the mentioned tool has a step-wise approach that find the next possible behavior at a time. In Chapter 6, we present our approach in details.

  Our work is the first tool support for priority in Reo. Chapter 7 elaborates on our approach in obtaining a priority-aware formal semantics of Reo from the solutions of constraints generated from each of Reo elements in a compositional manner.

4

Formal Semantics for Reo

4.1 Introduction

A benefit of employing coordination languages in general and Reo in particular is that they express the coordination patterns explicitly and separate them from the computational part of the code. This opens up possibilities for performing various types of analysis and automation such as model checking, code generation, automated test generation, etc.

To be able to perform such tasks, it is insufficient to describe the behavior of Reo models in a verbal manner. We need a more rigorous way to unambiguously specify semantics of Reo models.

Several formal semantics have been proposed in the recent years that express the behavior of Reo connectors. Jongmans et al. [JA12] present a comprehensive overview of thirty models. They grouped these models into the following categories:

- *Coalgebraic models*: Two coalgebraic semantics of Reo, Timed Data Streams [Arb02] [AR02] [RKNP04] and Record Streams [IB08] [IBC11] rely on the coalgebraic concept of *stream*, which refers to an infinite sequence of elements of a given set. This class of semantics are difficult to use for analysis purpose, for instance, as an underlying model to apply model checking techniques [JA12].

- *Automata-based semantics*: A big number of Reo operational semantics are based on automata. States in these automata correspond to the states of a Reo network, while the transitions denote I/O operations.

  A list of automata based semantics for Reo are: port automata (PA) [KC09], Constraint Automata [BSAR06], Labeled Constraint Automata (LCA) [KB09], Timed Constraint Automata (TCA) [ABBR04], Probabilistic Constraint Automata [Bai05], Quantitative Constraint Automata (QCA) [ACMM07] [MA09], Continuous Time Constraint Automata (CTCA) [BW06], Resource Sensitive Timed Constraint Automata (RSTCA) [MA07a], Transactional Constraint Automata (TNCA) [MA10], Behavioral Automata (BA) [Pro11], Buchi Automata [IB08] [IBC11] [IBC08] [IBC11], Guarded Automata [BCS12] [Mar09], Stochastic Guarded Automata [MSKA10] [MSKA14], Intentional Automata [Cos10], Quantitative Intentional Automata [ACvdM$^+$09], and Action Constraint Automata [KCA10].

- *Structural operational semantics*: Some of the semantics proposed for Reo are expressed in terms of structural operational semantics. Sun Meng et al. [MAA$^+$12] model Reo networks in terms of the Unifying Theories of Programming (UTP) [Hoa13]. A UTP design consists of predicates that express assumptions on inputs and commitments on outputs.

  Another work in this field is done by Mousavi et al. [MSA06]. They present a Structural Operational Semantics (SOS) for some of Reo primitives in Gordon Plotkin's style [Plo04]. In the proposed semantics, data-flow of a Reo connector is represented by a set of rules, which pair the structure of the connector with functions that map the nodes to potentially infinite sequences of data items.

  Tile Model [ABC$^+$09] is a more recent SOS-based formal semantics for Reo that extends Gordon Plotkin's SOS inference rules. In this model, transitions are described as movements from an initial state to a final state upon firing related triggers.

  Tile Model defines composition in three ways:

  - horizontal composition that models synchronization, where the effect of one tile is a trigger for another tile,
  - vertical composition, which is a composition occurring in time. This is when the final state of one tile matches the initial state of another tile,

- parallel composition that captures concurrency.

- *Semantics based on graph-coloring.* Connector coloring (CC) [CCA07] is a formal semantics for Reo that describes the behavior of a connector by assigning different colors to its ports.

  The colors designate presence or absence of data-flow. This model accounts for synchronization and context dependency. It captures context dependency by propagating negative information about the absence of data-flow inside a Reo network.

The most important types of semantics that have influenced and provided basis for the other classes of semantics are constraint automata and coloring semantics. These models are the underlying models of several tools for Reo ranging from animation to testing and model checking.

In this chapter, we present the definition and examples for Reo semantics that are relevant to this thesis. In addition, we briefly discuss the time complexity of obtaining formal semantics of a Reo network using the computation rules defined by the formal semantics.

4.2 Constraint automata

Definition 4.2.1 (Constraint automaton [BSAR06]) *A constraint automaton is a tuple $\mathcal{A} = (Q,\ \mathcal{N}, \to, q_0)$, where*

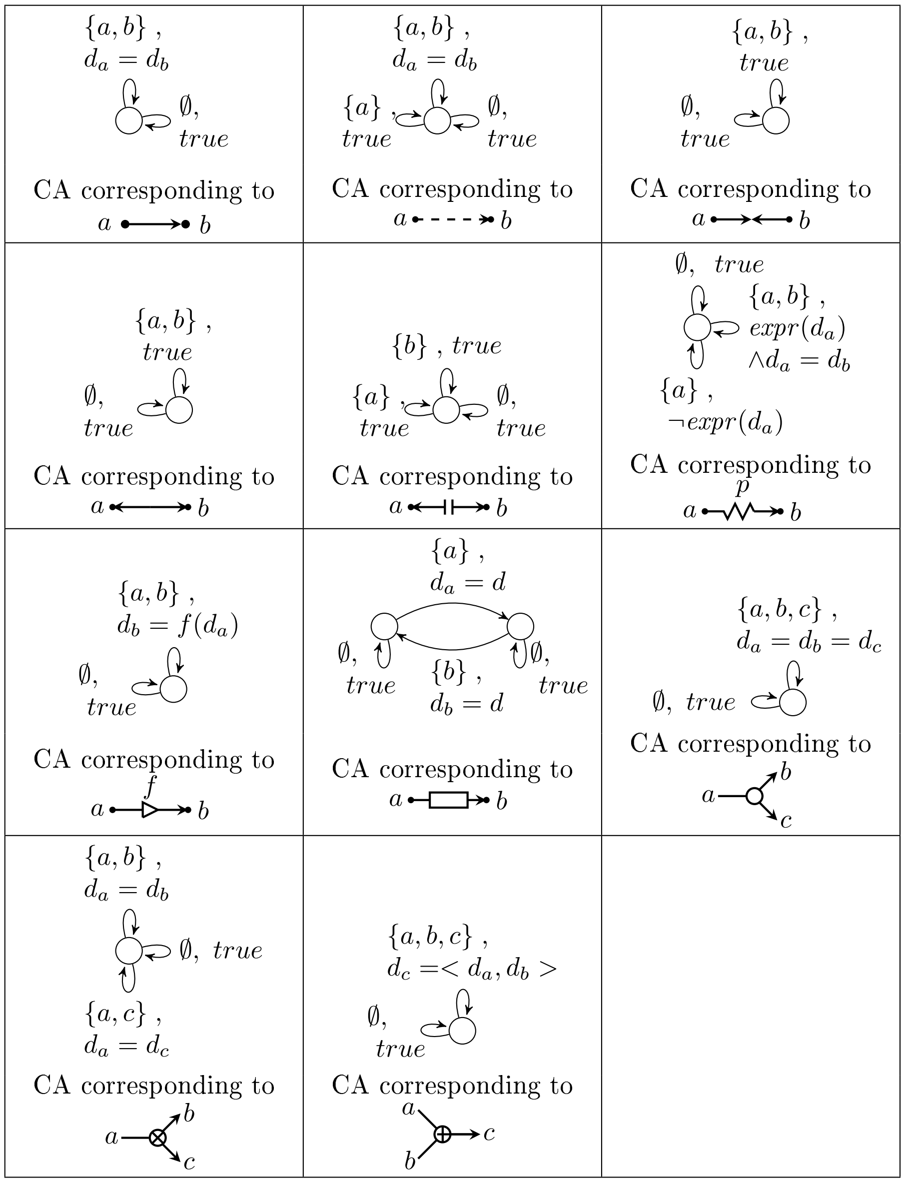
- *Q is a set of states,*
- *$\mathcal{N}$ is a set of port names,*
- *$\to \subseteq Q \times 2^{\mathcal{N}} \times DC \times Q$ is a transition relation, where DC is the set of data constraints over a finite data domain Data,*
- *$q_0 \in Q$ is an initial state.*

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \to$. Table 4.2.1 depicts the CA corresponding to the most common Reo elements.

Constraint automata have a compositional nature. Therefore, the semantics of a whole model can be obtained through the *composition* of the given semantics of its participant elements.

Following is the definition of the *product* operator, which performs the composition.

Table 4.2.1: Constraint automata for basic Reo primitives



Definition 4.2.2 (Product on constraint automata) *The product of constraint automata $\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \rightarrow_1, q_{0,1})$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \rightarrow_2, q_{0,2})$ is defined as:*

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

where the following rules define the transition relation $\rightarrow$:

$$\frac{q_1 \xrightarrow{N_1, g_1} p_1, q_2 \xrightarrow{N_2, g_2} p_2, N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{< q_1, q_2 > \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} < p_1, p_2 >}$$

$$\frac{q_1 \xrightarrow{N_1, g_1} p_1, N_1 \cap \mathcal{N}_2 = \emptyset}{< q_1, q_2 > \xrightarrow{N_1, g_1} < p_1, q_2 >}$$

$$\frac{q_2 \xrightarrow{N_2, g_2} p_2, \mathcal{N}_1 \cap N_2 = \emptyset}{< q_1, q_1 > \xrightarrow{N_2, g_2} < q_1, q_2 >}$$

We can abstract from the data-flow on certain Reo nodes using the *hiding* operator defined as follows:

Definition 4.2.3 (Hiding on constraint automata) *Let $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0)$ be a CA and $C \in \mathcal{N}$.*

The constraint automaton that results from hiding the node C in automaton $\mathcal{A}$ is $\exists C [\mathcal{A}] = (Q, \mathcal{N} \backslash \{C\}, \rightarrow_C, q_0)$ and the transition relation $\longrightarrow_C$ is defined as follows:

$$\frac{p \xrightarrow{N, g} q, N' = N \backslash \{C\}, g' = \exists C [g]}{p \xrightarrow{N', g'}_C q}, \text{ where}$$

$$\exists C [g] = \bigvee_{d \in \mathcal{D}} g [d(C)/d].$$

Example 4.2.1 *Figure 4.2.2 depicts the CA semantics of the Reo network of Figure 4.2.1. According to CA, it is possible that the* lossySync *channel loses the incoming data in the state q, where the* FIFO_1 *channel is empty. This is an example of undesired behavior that is the result of the fact that CA is not a context-dependent semantics.*

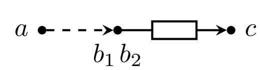


Figure 4.2.1: A context-dependent Reo connector

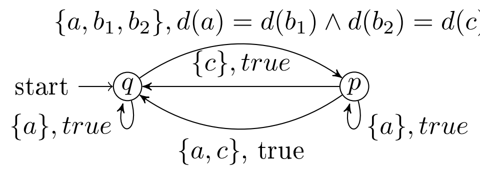


Figure 4.2.2: Constraint automaton of the Reo network of Figure 4.2.1

Example 4.2.2 *Figure 4.2.4 illustrates the CA of the Reo network of Figure 4.2.3. Since, CA is data-aware it can describes the correct behavior of this data-aware network.*

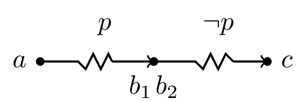


Figure 4.2.3: A data-aware Reo connector

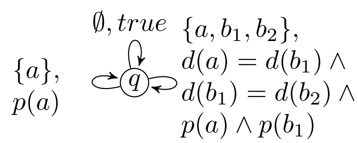


Figure 4.2.4: Constraint automaton of the Reo network of Figure 4.2.3

4.3 Constraint automata with state memory

Constraint automata with state memory (CASM) [PSHA12] extends CA with variables that represent local memory cells of automata states. Because CASM elaborates on state information, we choose to use CASM instead of CA, in our work.

Definition 4.3.1 (Constraint automaton with state memory) *A constraint automaton with state memory (CASM) is a tuple* $A = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ *where*

- Q *is a finite set of states.*
- $\mathcal{N}$ *is a finite set of names.*
- $\rightarrow$*, a finite subset of* $Q \times 2^{\mathcal{N}} \times DC(\mathcal{N}, \mathcal{M}, \mathcal{D}) \times Q$*, is the transition relation of* A*, where* $DC(\mathcal{N}, \mathcal{M}, \mathcal{D})$ *is the set of data constraints, defined below.*

- *$q_0 \in Q$ is an initial state.*
- *$\mathcal{M}$ is a set of memory cell names, where $\mathcal{N} \cap \mathcal{M} = \emptyset$.*

Every $n \in \mathcal{N}$ represents a node in a Reo connector. The set $\mathcal{N}$ is partitioned into three mutually disjoint sets of source nodes $\mathcal{N}^{src}$, mixed nodes $\mathcal{N}^{mix}$, and sink nodes $\mathcal{N}^{snk}$.

Because we make the replication and merge inherent in Reo nodes explicit as *replicator* and *merger* primitives, at most two primitive ends coincide on every node $n \in \mathcal{N}$. Thus, it follows that a source or a sink node contains only a single (source or sink) primitive end, and a mixed node contains exactly one source and one sink primitive ends.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \rightarrow$. For every transition $q \xrightarrow{N,g} p$, we require that $g \in DC(N, \mathcal{M}, \mathcal{D})$, where $\mathcal{D}$ is the global set of numerical data values and $DC(N, \mathcal{M}, \mathcal{D})$ is the language defined by the following grammar:

$$g \;::=\; \mathit{true} \mid \neg\, g \mid g \wedge g \mid u = u \mid u < u,$$

$$u \;::=\; d(n) \mid m' \mid m \mid v.$$

In this grammar,

- $=$ is the symmetric equality relation,
- $<$ is a total order relation,
- $n \in N \subseteq \mathcal{N}$ denotes a node name,
- $d(n)$ represents the data item exchanged through the node n,
- $m \in \mathcal{M}$ correspond to a memory cell in the current state, which is the source state of the transition,
- m' stands for the memory cell $m \in \mathcal{M}$ in the next state, which is the target state of the transition,
- $v \in \mathcal{D}$.

As usual, *false* stands for $\neg$*true*, $x > y$ stands for $y < x$, and other logical operators, such as $\vee$ and $\Rightarrow$ (the implication symbol) can be built from the given operators.

Transitions with data constraints that can be reduced to *false* using the Boolean laws are impossible and we omit them. A data constraint g that is always *true* can be left out.

We use $\mathcal{M}_g$ to represent the set of all $m \in \mathcal{M}$ that syntactically appear as m in a data constraint g; and $\mathcal{M}'_g$ to refer to the set of all $m \in \mathcal{M}$ that syntactically appear as m' in g.

The valuation function $\mathcal{V}_q : \mathcal{M} \rightarrow 2^{\mathcal{D}}$ designates the set of values $\mathcal{V}_q(m)$ of a memory cell $m \in \mathcal{M}$ in a state $q \in Q$, where $\mathcal{V}_{q_0}(m) = \emptyset$ for all $m \in \mathcal{M}$.

A transition $q \xrightarrow{N,g} p$ in a given constraint automaton with state memory is possible only if there exists a substitution for every syntactic element $d(n)$, m, and m' that appears in g to satisfy g.

A substitution simultaneously replaces in g:

- every occurrence of $d(n)$ with the data value exchanged through the node $n \in \mathcal{N}$;

- every occurrence of m' of every $m \in \mathcal{M}$ with a value $v \in \mathcal{D}$;

- every occurrence $m \in \mathcal{M}$ with:

  – the special symbol $'\circ'$ if $\mathcal{V}_q(m) = \emptyset$,

  – a value $v \in \mathcal{V}_q(m)$, otherwise.

The guard g is satisfied if proper replacement values can be found to make g *true*. Making this transition, the automaton defines the valuation function $\mathcal{V}_p$ for the target state p, as follows:

- For every $m \in \mathcal{M}'_g$, $\mathcal{V}_p(m)$ is the set of all $v \in \mathcal{D}$ whose replacements for m' satisfy g.

- For every other $m \in \mathcal{M}$, $\mathcal{V}_p(m) = \emptyset$.

A relational operator evaluates to *true* only if the values of its operands are in its respective relation. Thus, any operator with one or more $\circ$ as an operand always evaluates to *false*.

We call a CASM, *normalized* iff

- It does not have two states with the same set of state memory variables.

- Every two transitions differ at least in their start states, their target states, or their sets of synchronizing ports.

For any arbitrary CASM that is not normalized, we can normalize it by

- introducing auxiliary variables, to make the set of state memory variables unique for each state,

- by merging the transitions that have the same start and target states and synchronize the same ports.

In the sequel, we consider only *normalized* CASMs.

Following are the definitions for *product* and *hiding* operations on CASM. Both definitions are adapted from [BSAR06].

Definition 4.3.2 (Product automaton on CASM) *The product of CASMs* $\mathcal{A}_1 = (Q_1,\ \mathcal{N}_1,\ \to_1,\ q_{0,1},\ \mathcal{M}_1)$ *and* $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \to_2, q_{0,2}, \mathcal{M}_2)$ *is defined as:*

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \to, q_{0,1} \times q_{0,2}, \mathcal{M}_1 \cup \mathcal{M}_2)$$

where the following rules define the transition relation $\to$*:*

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

$$\frac{q_1 \xrightarrow{N_1,g_1}{}_1 p_1, N_1 \cap \mathcal{N}_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N_1,g_1} \langle p_1, q_2 \rangle} \qquad \frac{q_2 \xrightarrow{N_2,g_2}{}_2 p_2, \mathcal{N}_1 \cap N_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N_2,g_2} \langle q_1, p_2 \rangle}$$

Similar to CA, we can abstract from the data-flow on certain Reo nodes using the *hiding* operator defined as follows:

Definition 4.3.3 (Hiding on CASM) *Let* $\mathcal{A} = (Q, \mathcal{N},\ \to, q_0, \mathcal{M})$ *be a CASM and* $C \in \mathcal{N}$*.*

The constraint automaton that results from hiding the node C *in automaton* $\mathcal{A}$ *is* $\exists C\,[\mathcal{A}] = (Q, \mathcal{N}\backslash\{C\},\ \to_C, q_0, \mathcal{M})$ *and the transition relation* $\longrightarrow_C$ *is defined as follows:*

$$\frac{p \xrightarrow{N,g} q, N' = N\backslash\{C\},\ g' = \exists C\,[g]}{p \xrightarrow{N',g'}{}_C\ q},\ \textit{where}$$

$$\exists C\,[g] = \bigvee_{d \in \mathcal{D}} g\,[d\,(C)\,/d]\,.$$

To facilitate our further reasoning with CASM, we provide the following definition that gives the set of state memories used in each state.

Definition 4.3.4 (State variables) *Given the CASM* $\mathcal{A} = (Q, \mathcal{N},\ \to, q_0, \mathcal{M})$*, we define the function* $S : Q \to 2^{\mathcal{M}}$ *as for each* $q \xrightarrow{N,g} p$*,* $m \in V_g \Rightarrow m \in S\,(q)$ *and* $m' \in V_g \Rightarrow m \in S\,(p)$*.*

Example 4.3.1 *Figure 4.3.2 depicts the CASM for the Reo shown network in Figure 4.3.1. CASM provides an explicit representation for the stored values using its state variables.*

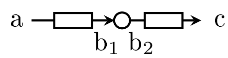


Figure 4.3.1: FIFO_2

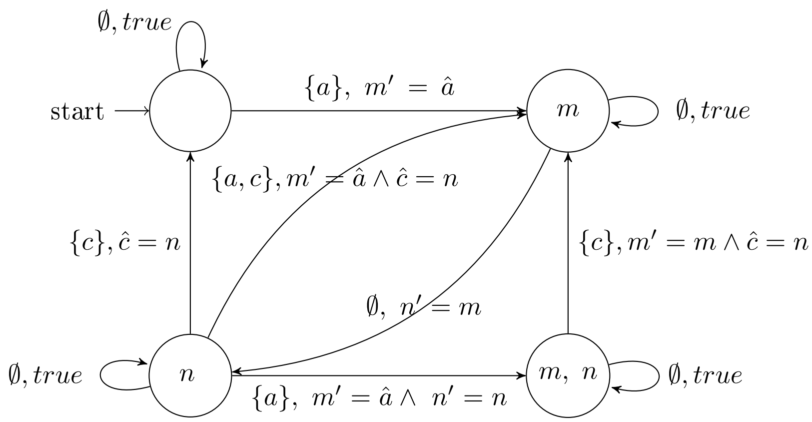


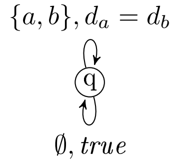
Figure 4.3.2: Constraint automaton of the Reo network of Figure 4.3.1

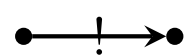
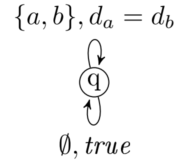
4.4 Constraint automata with priority

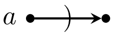
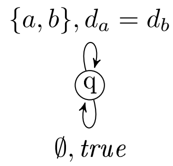
Definition 4.4.1 (Constraint automaton with priority) *A constraint automaton with priority is a tuple* $\mathcal{P} = (\mathcal{A}, \mathcal{R}, \mathcal{S}, \mathcal{T})$ *where*

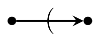
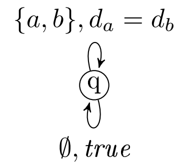
- $\mathcal{A} = (\mathcal{Q}, \mathcal{N}, \mathcal{N}^{mix}, \mathcal{N}^{src}, \mathcal{N}^{snk}, \longrightarrow, \mathcal{Q}_0)$ *is a constraint automaton,*
- $\mathcal{R} \subset 2^{\mathcal{N}} : \forall R \in \mathcal{R}$ *is a subset of* $\mathcal{N}$*, such that if a node* $n \in R$ *connects to the priority imposing channel, PrioritySync, the priority affects* $\bar{n} \in R$*.*
- $S \subset \mathcal{R} \times \mathcal{R}$ *is the set pairs of subsets of* $\mathcal{N}$*, such that* $\forall (X, Y) \in S$*, the priority imposed on the region* X *can propagate to the region* Y*,*
- $\mathcal{T} =^{def} (t, \triangleleft) : t \in R$ *and* $\triangleleft \subseteq \longrightarrow \times \longrightarrow$ *is a binary relation on the transitions of* A *such that* $q \xrightarrow{N,g} p \triangleleft \bar{q} \xrightarrow{\bar{N},\bar{g}} \bar{p}$ *implies* $q = \bar{q}$ *and* $(N, g) \neq (\bar{N}, \bar{g})$*.*

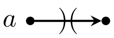
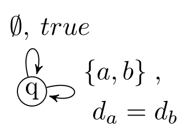
Table 4.4.1: Priority constraint automata of commonly used Reo primitives

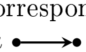
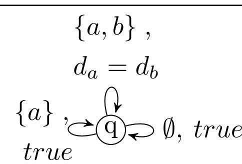
<table>
<tr>
<td>


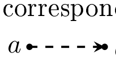
$Q_0 = \{q\},$
$R = \{\{a,b\}\},$
$S = 1,$
$T = \{\{a,b\}:$
$q \xrightarrow{\{a,b\},d_a=d_B} q \lhd q \xrightarrow{\emptyset,true} q\}$
CAP corresponding to
a  b
</td>
<td>


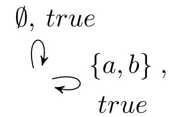
$Q_0 = \{q\},$
$R = \{\{a\},\{b\}\},$
$S = 1 \cup \{(\{b\},\{a\})\},$
$T = \emptyset$
CAP corresponding to
a  b
</td>
</tr>
<tr>
<td>


$Q_0 = \{q\},$
$R = \{\{a,b\}\},$
$S = 1 \cup \{(\{a\},\{b\})\},$
$T = \emptyset$
CAP corresponding to
a  b
</td>
<td>


$Q_0 = \{q\},$
$R = \{\{a\},\{b\}\},$
$S = 1 \cup \{(\{a\},\{b\}),(\{b\},\{a\})\},$
$T = \emptyset$
CAP corresponding to
a  b
</td>
</tr>
<tr>
<td>


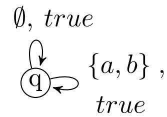
$Q_0 = \{q\},$
$R = \{\{a,b\}\},$
$S = 1,$
$T = \emptyset$
CAP corresponding to
a  b
</td>
<td>


$Q_0 = \{q\},$
$R = \{\{a,b\}\},$
$S = 1,$
$T = \{\emptyset : q \xrightarrow{\{a,b\},d_a=d_b} q \lhd q \xrightarrow{\{a\},true} q,$
$\emptyset : q \xrightarrow{\{a,b\},d_a=d_b} q \lhd q \xrightarrow{\emptyset,true} q,$
$\emptyset : q \xrightarrow{\{a\},true} q, \lhd q \xrightarrow{\emptyset,true} q\}$
CAP corresponding to
a  b
</td>
</tr>
</table>



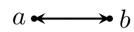
$Q_0 = \{q\}$,
$R = \{\{a, b\}\}$,
$S = 1$,
$T = \emptyset$

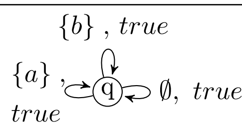
CAP corresponding to



$Q_0 = \{q\}$,
$R = \{\{a, b\}\}$,
$S = 1$,
$T = \emptyset$

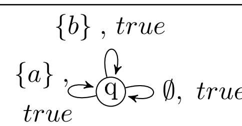
CAP corresponding to





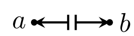
$Q_0 = \{q\}$,
$R = \{\{a, b\}\}$,
$S = 1$,
$T = \emptyset$

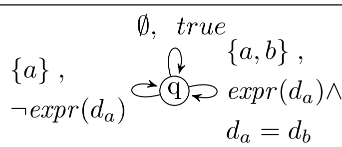
CAP corresponding to



$Q_0 = \{q\}$,
$R = \{\{a, b\}\}$,
$S = 1$,
$T = \emptyset$

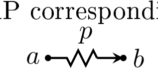
CAP corresponding to

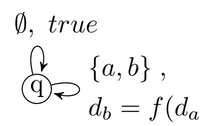




$Q_0 = \{q\}, R = \{\{a, b\}\}$,
$S = 1, T = \emptyset$

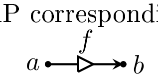
CAP corresponding to

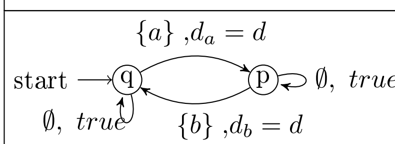




$Q_0 = \{q\}, R = \{\{a, b\}\}$,
$S = 1, T = \emptyset$

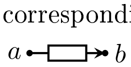
CAP corresponding to

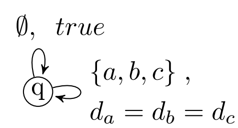




$Q_0 = \{q\},\ R = \{\{a\}, \{b\}\}$,
$S = 1,\ T = \emptyset$

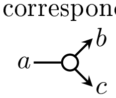
CAP corresponding to

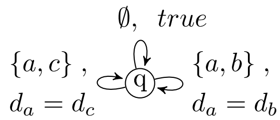
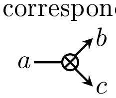
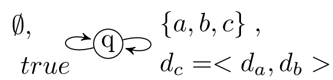
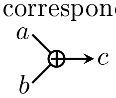




$Q_0 = \{q\}, R = \{\{a, b, c\}\}$,
$S = 1,\ T = \emptyset$

CAP corresponding to



 $Q_0 = \{q\}$, $R = \{\{a,b,c\}\}$, $S = 1$, $T = \emptyset$ CAP corresponding to 	 $Q_0 = \{q\}$, $R = \{\{a,b,c\}\}$, $S = 1$, $T = \emptyset$ CAP corresponding to 

Observe that the nodes in R connect to each other by priority propagating channels such as Sync, PrioritySync, SyncDrain. The connections of the regions paired in S is, however, via priority blocking channels like BlocingSinkSync, BlockingSourceSync and AsyncDrain. The sets $\mathcal{R}$, $\mathcal{S}$ and the tag t in $\mathcal{T}$ are auxiliary concepts for composition of CAPs. Table 4.4.1 shows CAPs corresponding to Reo elements.

Similar to CA, the *product-automaton* operator ($\bowtie$) computes the CAP corresponding to a Reo network from CAPs of its substituent elements.

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be the two CAPs, $\tau_1, \lambda_1 \in \longrightarrow_1$, $\tau_1 \lhd \lambda_1$ and $\tau_2, \lambda_2 \in \longrightarrow_2$. If τ_1 and τ_2 synchronize to form a transition $\tau \in \longrightarrow_{P_1 \bowtie P_2}$, λ_1 and λ_2 synchronize to form a transition $\lambda \in \longrightarrow_{P_1 \bowtie P_2}$, the relation of $\tau \lhd \lambda$ is *full lifting* of the $\tau_1 \lhd \lambda_1$.

Since the priority blocking channels can affect the propagation of the priority, the priority relations that full lifting defines are not always valid on the product of the automata. We need to eliminate invalid transitions that are results of improper propagation of the priority.

The following three cases are the only valid propagation of the priority [ABS15]:

- *Propagation over empty transitions*: If λ is an empty transition, then λ_1 and λ_2 are also empty transitions. In this case, full lifting brings a new priority imposition as: $\tau \lhd \lambda$.

- *Propagation by containment*: If λ_1 is a proper transition, then λ is a proper transitions, which contains λ_1. Therefore, full lifting is a natural growth of the previously imposed priority that preserves the priority relation as: $\tau \lhd \lambda$.

- *Propagation by seepage*: If λ_1 is an empty transition, but λ is a proper transition, then λ_2 is also a proper transition. Under this condition, full lifting is not always valid. Therefore, we need more restriction to preserve the new priority relation that full lifting impose that is $\tau \lhd \lambda$. The seepage relation $\mathcal{S}$ and the tag t of the transition help to check the validity of full lifting for this case. So, the full lifting is valid if there exists a finite sequence of regions $r_0, .., r_i, r_{i+1}, .., r_n$ such that $r_i \in R, (r_i, r_{i+1}) \in \mathcal{S}, r_0 = t$ and r_n includes all nodes involved in the transition λ_2. Note that $\mathcal{S}$ is the seepage relation that defines the allowed propagation of the priority through regions. Observe that if $t_1 = \emptyset$, then $t = \emptyset$. Since $\emptyset \notin \mathcal{R}$, such a sequence does not exist and the full lifting is not valid.

Following is the definition of the CAP product operator.

Definition 4.4.2 (Product-automaton) *Let $\mathcal{P}_i = (\mathcal{A}_i, \mathcal{R}_i, \mathcal{S}_i, \mathcal{T}_i)$, $i = 1, 2$ be two CAPs, where $\mathcal{A}_i = (\mathcal{Q}_i, \mathcal{N}_i, \mathcal{N}_i^{mix}, \mathcal{N}_i^{src}, \mathcal{N}_i^{snk}, \longrightarrow, \mathcal{Q}_{0,i})$, such that:*

$$\mathcal{N}_1 \cap \mathcal{N}_2 \subseteq \mathcal{N}_1^{src} \cap \mathcal{N}_2^{snk} \cup \mathcal{N}_1^{snk} \cup \mathcal{N}_2^{src}$$

The definition of the product-automaton $\mathcal{P}_1 \bowtie \mathcal{P}_2 = (\mathcal{A}_1 \bowtie \mathcal{A}_2, \mathcal{R}, \mathcal{S}, \mathcal{T})$ follows:

Listing 4.1: Calculating $\mathcal{R}$

```
𝓡 := ∅
for each  r₁ ∈ 𝓡₁
if ∃ r₂ ∈ R₂ : r₁ ∩ r₂ ≠ ∅
   𝓡 := 𝓡 ∪ r₁ ∪ r₂
 else
   𝓡 := 𝓡 ∪ r₁
for each  r₂ ∈ 𝓡₂
    if  ∄r₁ ∈ R₁ : r₁ ∩ r₂ ≠ ∅ then
       𝓡 = 𝓡 ∪ r₂
```

Listing 4.2: Calculating seepage relation $\mathcal{S}$

```
𝓢 := ∅
for each (u₁, v₁) ∈ 𝓢₁
  𝓢 += (big(u₁), big(v₁))
  for each (u₂, v₂) ∈ 𝓢₂
     𝓢 += (big(u₂), big(v₂))
    𝓢 += 𝓘
```

Let $(t_1, \tau_1 \lhd_1 \lambda_1) \in \mathcal{T}_1$. The transition λ_1 is either empty or proper:

$$\begin{array}{ll} \forall\, \tau_2 \ \in \longrightarrow_2 : \ \tau_1 \cap \tau_2 \neq \emptyset & \text{if } \lambda_1 \text{is empty} \\ big(r_1) \ : \ \tau_1 \mid\mid \tau_2 \ \lhd \ \emptyset & \\ \forall\, \tau_2 \ \in \longrightarrow_2 : \ \tau_1 \cap \tau_2 = \emptyset & \\ if \ exists \ a \ sequence \ such \ that & \text{otherwise} \\ \forall\, \tau_2 \ : \ \tau_1 \cap \tau_2 \neq \emptyset & \lambda_1 \text{ is proper} \\ \forall\, \lambda_2 \ : \ \lambda_1 \cap \lambda_2 \neq \emptyset & \\ big(r_1) \ : \ \tau_1 \mid\mid \tau_2 \ \lhd \ \lambda_1 \mid\mid \lambda_2 & \end{array} \tag{4.1}$$

4.5 Connector coloring

The connector coloring semantics [CCA07] denote the existence or absence of data-flow through the primitive ends by marking them with different colors.

Let $\mathcal{C}olors$ be a set of colors. In a set of two colors, $\mathcal{C}olors = \{—, \text{- -}\}$, — denotes an occurrence and - - represents an absence of data-flow. Two colors are adequate to express the formal semantics of many Reo networks. However, they cannot express the semantics of context-dependent Reo networks.

Such a network presented in Example 4.2.2 is when the sink end of a *lossySync* channel connects to an empty $FIFO_1$ channel; in this case, the semantics of this network according to the two-color set includes the case where the *lossySync* loses its incoming data item, while the $FIFO_1$ channel is empty. This is an unacceptable behavior for a so-called context-dependent *lossySync* channel: it must lose its incoming data only if its sink end cannot dispense it. In the sequel, when we refer to a *lossySync* we mean its context sensitive version.

The three coloring semantics, $\mathcal{C}olors = \{-, \lhd, \rhd\}$, addresses this problem by propagating negative information regarding the absence of data-flow. It replaces - - with $\lhd$ and $\rhd$ meaning that the associated primitive end, respectively, *provides* or *requires* a reason for no-flow.

Considering that no-flow can occur only when at least one of the involved primitive ends *provides* a reason for it, and that an empty $FIFO_1$ cannot *provide* a reason for no-flow on its source end, the invalid behavior described above does not arise in the three coloring semantics.

Definition 4.5.1 (Coloring) *A coloring $l : \mathcal{P} \to \mathcal{C}olors$ is a total function from the primitive ends to a set of colors. We refer to the global set of colorings as $\mathcal{L}$.*

Definition 4.5.2 (Coloring composition) *The composition of colorings l_1 and l_2, denoted $l_1 \bullet l_2$, is defined as:*

$$
\begin{aligned}
l_1 \bullet l_2 = \{ & \\
& c_1 \cup c_2 | c_1 \in l_1, c_2 \in l_2, p_1 \in dom(c_1), p_2 \in dom(c_2), \\
& p_1 \textit{ and } p_2 \textit{ are the source and sink ends of a node } n, \\
& \neg\ (\ c_1(p_1) = \triangleleft\ \wedge\ c_2(p_2) = \triangleright\ ) \\
\} &
\end{aligned}
$$

Definition 4.5.3 (Coloring table) *A coloring table over the primitive set $P \subseteq \mathcal{P}$ is a set of colorings with the domain P.*

Definition 4.5.4 (Next function) *The next function $\eta : \mathcal{L} \times 2^{\mathcal{L}} \rightarrow 2^{\mathcal{L}}$ maps a pair of a coloring and a coloring table to a colorings table.*

Definition 4.5.5 (Coloring semantics) *A coloring semantics of a Reo network is a tuple $CC = \langle \mathcal{P}, 2^{\mathcal{L}}, l_0, \eta \rangle$, where:*

- *$\mathcal{P}$ is the set of primitive ends,*
- *$l_0 \in \mathcal{L}$ is the initial set of possible colorings,*
- *$2^{\mathcal{L}}$ is a set of colorings,*
- *η is a next function that maps a pair of a coloring and a coloring table into a coloring table.*

Example 4.5.1 *Table 4.5.1 depicts the CC for the network shown in Figure 4.5.1. The two flows described in the table correspond to the cases; i) when there is a write request of the end a, then the ends a, b_1 and b_2 have a flow, but the end c provides a reason for no flow, ii) when there is no write request present on the end a, therefore the ends a and b_2 require a reason for no flow and the ends b_1 and c provides a reason for no flow. Since CC is context-sensitive, it can capture the semantics of the given network correctly.*

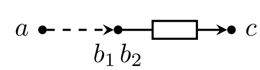


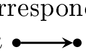
Figure 4.5.1: A context-dependent Reo connector

Table 4.5.1: Connector coloring semantics of the Reo network of Figure 4.5.1

a	b_1	b_2	c
—	—	—	$\triangleright$
$\triangleright$	$\triangleright$	$\triangleright$	$\triangleright$

Table 4.5.2: Connector coloring semantics of commonly used Reo primitives

a	b
×	×
○	●
●	○

CC corresponding to
a  b

a	b
×	×
○	×
×	○

CC corresponding to
a b

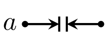
a	b
×	×
○	●
●	○

CC correspondence to
a b

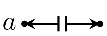
a	b
×	×
○	●
●	○

CC correspondence to
a b

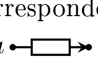
a	b
×	×
○	●
●	○

CC correspondence to
a  b

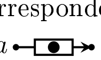
a	b
×	×
○	●
●	○

CC correspondence to
a  b

a	b
×	×
○	●
●	○

CC correspondence to
a  b

a	b
×	×
○	●
●	○

CC correspondence to
a  b

Example 4.5.2 *Table 4.5.3 shows the CC of the Reo network shown in Figure 4.5.2. The absence of data constraints in the CC, leads to incorrect behavior, as shown in the first row of the table, where there is flow on both b_1 and c.*

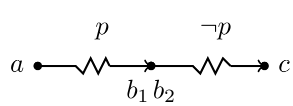


Figure 4.5.2: A data-aware Reo connector

Table 4.5.3: Connector coloring semantics of the Reo network of Figure 4.5.2

a	b_1	b_2	c
—	—	—	—
—	—	—	$\triangleright$
—	$\triangleright$	$\triangleright$	$\triangleright$
$\triangleright$	$\triangleright$	$\triangleright$	$\triangleright$

4.6 Reo automata

Bonsangue et al. [BCS12] present *Reo automata* (RA), an automata-based formal model, to deal with context-dependency in Reo.

Intuitively, a Reo automaton is a non-deterministic automaton whose transitions are labeled in the form of $g|f$, where g is a binary predicate, called *guard*, and f a set of nodes that fire synchronously. A transition can be taken only when its guard g is true.

Let $\Sigma = \{\sigma_1, ..., \sigma_k\}$ be a set of nodes, $\bar{\sigma}$ be the negation of σ, and $\mathcal{B}_\Sigma$ be the free Boolean algebra generated by the following grammar:

$$g \ ::= \ \sigma \ \in \ \Sigma \mid \top \mid \bot \mid g \ \vee \ g \mid g \ \wedge \ g \mid \bar{g}$$

The above grammar produces *guards*. Often $g_1 \wedge g_2$ is written as $g_1 g_2$. A natural order $\leq$ is defined between two guards g_1,$g_2 \in \mathcal{B}_\Sigma$ as

$$g_1 \leq g_2 \Rightarrow g_1 \wedge g_2 = g_1$$

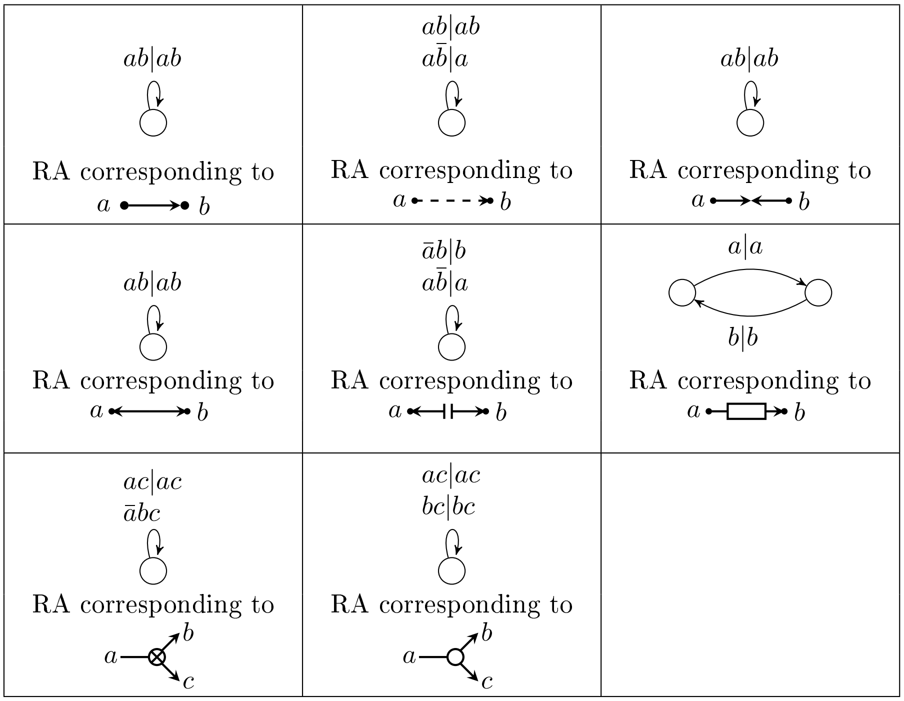
The intended interpretation of $\leq$ is logical implication: $g_1 \implies g_2$. An atom of $\mathcal{B}_\Sigma$ is a guard $a_1...a_k$ such that $a_i \in \bar{Sigma} \cup \Sigma$ with

$$\Sigma = \{\sigma_i \mid \sigma_i \in \Sigma\}, 1 \leq i \leq k$$

Definition 4.6.1 (Reo automaton [BCS12]) *A Reo automaton is a triple* (Σ, Q, δ) *where:*

- Σ *is the set of nodes,*
- Q *is a set of states,*

Table 4.6.1: Reo automata for basic Reo primitives



- *$\delta \subseteq Q \times \mathcal{B}_\Sigma \times 2^\Sigma \times Q$ is the transition relation such that for transitions labeled as $\mathcal{B}_\Sigma \times 2^\Sigma$ such that for each $q \xrightarrow{g|f} p \in \delta$:*
  - $g \ \leq \hat{f}$
  - $g \ \leq g' \ \leq \hat{f}.\ \forall \alpha \leq \ g'.\ \exists\ q \xrightarrow{g''|f} p \in \Sigma.\ \alpha \leq g''$

Table 4.6.1 depicts the Reo automata corresponding to the most common Reo elements.

4.7 Complexity

Analyzing the complexity of the calculations on CAP or other formal semantics of a Reo network in a formal fashion is beyond the scope of this dissertation. However, here we roughly estimate the time complexity of the product of CA. We have chosen CA because it is one of the most basic formal semantics for Reo. Calculating the complexity of CA product can provide an insight into the complexity of composing more sophisticated automata based semantics such as CAP.

Let R be a Reo network that is constructed by connecting n smaller networks in a step-wise fashion, meaning that one join occurs at a time, $\mathcal{A}_{1..i-1} = (Q_{1..i-1}, \mathcal{N}_{1..i-1}, \rightarrow_{1..i-1}, q_{0_{1..i-1}})$ be the CA of $R_{1..i-1}$ network at the i-th step before the i-th network is added, and $\mathcal{A}_i = (Q_i, \mathcal{N}_i, \rightarrow_i, q_{0_i})$ be the CA of R_i, the i-th network.

Note that at the first step, only A_1 exists. At the second step A_1 is connected to A_2 to form $A_{1..2}$.

Computing $\mathcal{A}_{1..i-1} \bowtie \mathcal{A}_i$ requires all transitions of $\mathcal{A}_{1..i-1}$, $t_{1..i-1}$, to be checked against the transitions of $\mathcal{A}_i$, t_i. For each t_i, the common ports of the transition and $\mathcal{N}_{1..i-1}$ need to be found. The time complexity of this operation is $O(T_{1..i-1} \times P_{1..i-1} \times P_i)$, where $T_{1..i-1}$ is the number of transitions of $\mathcal{A}_{1..i-1}$, $P_{1..i-1}$, and P_i are the number of elements in $\mathcal{N}_{1..i-1}$ and $\mathcal{N}_i$, respectively.

In addition, for the each $t_{1..i-1}$ all the common ports of the transition with $\mathcal{N}_i$ is calculated. With a similar complexity of $O(T_i \times P_{1..i-1} \times P_i)$, where T_i is the number of transitions of $\mathcal{A}_i$.

Based on the outcome of these operations, we may need to create a couple of new states by merging the source and target states of $t_{1..i-1}$ and t_i. We assume that the creating these states takes a constant time. This assumption is based on the fact that constraint automata states are atomic entities.

However, in the case of CASM, the time complexity of creating a new state in the product of two CASMs depends on the number of state variables. Without considering transition guards, the complexity of computing $\mathcal{A}_1..i$ is:

$$O(T_{1..i-1} \times P_{1..i-1} \times P_i \ + \ T_i \times P_{1..i-1} \times P_i \ + \ T_{1..i-1} \times T_i) =$$

$$O(\prod_{j=1}^{i-1} T_j \times \prod_{k=1}^{i} P_k \ + \ \prod_{l=1}^{i} P_l \times T_i \ + \ \prod_{m=1}^{i} T_m)$$

Assuming that the number of transitions and the port names in each $\mathcal{A}_i$ is $\mathcal{T}$ and $\mathcal{P}$, respectively, the complexity can be written as $O(\mathcal{T}^n \times \mathcal{P}^n)$. As the formula shows the CA product is a very computationally expensive operation.

The problem of solving transition guards is a constraint satisfaction problem, which is a known NP-Complete problem. It is known that verifying a solution to an NP-complete problem is possible in polynomial time, but the time to find the solutions increases rapidly by the growth in the size of constraints.

Later in this dissertation, we provide an alternative approach for obtaining the formal semantics of a Reo network using constraint solvers. Our approach enables us to benefit from all the advances in research to keep this problem tractable for practical use.

5

Mapping BPMN to Reo

In this chapter, we present our approach in transforming BPMN 2 models into Reo networks. Since the core of Extensible Coordination Tool-set (ECT) [AKM+08a] and Eclipse BPMN 2 modeler [act] are based on Eclipse Modeling Framework (EMF) [SBPM09], the BPMN 2 to Reo transformation can be carried out in the model-driven paradigm. We use the Eclipse de-facto model transformation language and toolkit called Atlas Transformation Language (ATL) [JK05].

ATL is a high level rule-based language dedicated to model transformation. By using ATL we benefit from the power of separation of concerns and focus only on the required mapping rules, rather than matching patterns on the source models and execution of the rules.

The mapping rules presented in this chapter are mainly based on the conceptual mapping of BPMN primitives to Reo presented in [AKM08b] [AM08]. The following is a brief summary of the mapping:

- A task or a collapsed sub-process is mapped to a $FIFO_1$ channel, which denotes a unit of work in a process. However, an expanded sub-processes is modeled using a Reo connector whose inner elements are mapped from the inner elements of the sub-process.

- In general, an event is mapped to a replicator node. For each start event, a writer is created and connected to a source end of the node to simulate the arrival of the event. Similarly, each end event is connected to a reader on one of its sink ends. Throwing events are connected to the corresponding catching events using $FIFO_1$ and lossySync channels. So, they do not block the flow in case that the catching events are not yet ready to receive the event.
  - For each conditional event, a filter channel with the corresponding condition is created and connected to the source end of the node.
  - The terminate and throwing compensation are special cases, which their mappings requires possible compensations. Therefore, they have more sophisticated mappings, which we discuss in this chapter.
- Gateways are mapped to different kinds of Reo nodes based on their types and the number of their incoming and outgoing sequence flows.
  - A data-based exclusive gateway is mapped to a router node, while each of its outgoing sequence flows is mapped to a filter channel with a corresponding condition.
  - A data-based inclusive gateway is mapped to a replicator node.
  - A parallel event-based gateway with one incoming flow is mapped to a replicator. In case that it has more than one incoming flows, it is mapped to a join node.
- Sequence and message flows are mapped to synchronous channels unless there exists a more specific rule that describes the mapping in a given context.

Most BPMN 2 elements can be mapped to Reo constructs, which have relatively similar granularity. One notable exception is that mapping of transactions requires more effort than the other BPMN 2 elements do, and it creates many more Reo constructs. This is due to the complex behavior of BPMN 2 transactions compared to the other elements.

Tasks in a transaction should be compensated in the reverse order of their execution. In addition, the post compensation flow cannot be taken unless all performed compensatable tasks are compensated. Addressing these concerns requires more elements to be added to the target model.

Since for mapping transactions requires more work compared with the rest of elements. We refine them with groups of finer grained elements, which collectively deliver the same functionality. This is done prior to performing the transformation.

The rest of this chapter is organized as follows: Section 5.1 presents an algorithm to refine BPMN 2 transactions in order to simplify the mapping procedure. Section 5.2 is a brief introduction to Atlas Transformation Language (ATL). Our proposed BPMN 2 to Reo mapping is given in Section 5.3. We show result of the mapping using an example in Section 5.4. Section 5.5 overviews the related work on transformation of BPMN models.

5.1 Transaction refinement

To simplify mapping of BPMN transactions, we substitute them with a set of BPMN 2 elements that are easier to map to Reo, yet collectively expose the same functionality. The correctness of this refinement can be checked against the informal behavioral description of the elements involved. We do not provide a formal proof.

The mechanism to trigger a compensation in BPMN 2 is either by using a cancel event attached to the boundary of a transaction or by throwing a compensation event. For simplicity, we assume that all compensations are triggered in the former way. It is not a limiting assumption as it is possible to convert the latter to the former.

In the refinement process, we create complex gateways for two purposes: i) to control the execution order of compensation tasks and ii) to delay the post compensation flow. We refer to them as compensation order and post compensation, respectively.

We use these complex gateways as placeholders to be replaced by groups of Reo elements, which implement the informally described behavior of the gateways. Though the behavior of complex gateway is defined by its expression attribute, for these gateways, we ignore their expression attribute. During the refinement process, though, we keep track of these gateways and pass their identifiers to the ATL mapping process in order to invoke the suitable mapping rules.

We carry out the refinement as follows:

1. We create a send signal event for each compensatable task and place it after the task (using an inclusive gateway if the task has a following element). This is to notify when the task is completed.

2. When a compensatable task resides in a sequence of compensatable tasks, only the last performed task can be compensated immediately upon receiving the cancel event. The rest of the tasks should be compensated only if their

following tasks in the sequence are compensated. Therefore, for each compensatable task in a sequence except for the last task, we create a send signal event and place it after the compensation task corresponding to that task (using gateways for connecting objects when it is necessary). These events are fired after the corresponding compensatable tasks are compensated.

3. For a compensatable task T_a with a following compensatable task T_b in a sequence of compensatable tasks, we create a complex gateway (of type compensation order) with incoming sequence flows originating from 1) the cancel boundary event, 2) a newly created receive signal event, which catches the signal corresponding to completion of T_a, 3) a newly created receive signal event, which catches the signal corresponding to completion of T_b, and 4) a newly created receive signal event, which catches the signal corresponding to completion of the compensation of T_b. The complex gateway sends flow to the compensation task corresponding to T_a only if all incoming sequence flows are enabled.

The above steps assure that the compensation tasks are invoked in the right order. In addition, we need to prevent that the outgoing sequence flow of the cancel boundary event is taken before all compensation tasks within the given transaction are completed. The following step realizes this.

4. Let c_e be the cancel boundary event of the given transaction, s_e be the outgoing sequence flow of c_e, and f_e be the target of s_e. We create a new complex gateway g_e (of type post compensation) and remove s_e. For each compensation task t_c and its corresponding componsatable task t_a, we create a new receive signal event to receive these signals. For each event, we create a sequence flow, which has the event as its source and g_c as its target. This complex gateway enables its outgoing sequence flow if the cancel event is received and after receiving each receive signal event corresponding to the compensatable task t_a, the receive signal event corresponding to the compensation of the task t_c is received, as well.

Listings 5.1, 5.2, and 5.3 depict our algorithm for transaction refinement. To reduce verbosity, we provide the following definitions:

- The *objects* property of a transaction is the set of its enclosed BPMN 2 flow objects (i.e. activities, gateways, and events).
- The *compensation* property refers to the compensation task corresponding to the activity. If the task is not compensatable, this value is *null*.

- The *nextFlowObjects* property is the set of all the flow objects that are directly connected to an outgoing sequence flow from the flow object.

- The *previousFlowObjects* property is the set of all the flow objects that are directly connected to an incoming sequence flow from the flow object.

- The *receivers*, a property of a send signal event, is the set of the receivers of the event.

- The *getDoneSignal* function maps a compensatable or a compensation task to their corresponding send signal event.

- The *getNextCompensatables* function maps a compensatable task to its following compensatable tasks in sequences of compensatable tasks if they exist. Otherwise, it returns *null*.

In addition, we assume that adding an object to the nextFlowObjects list creates the required connecting objects.

The refinement starts with the refine method, which goes through the transactions in a given process and asserts that they have a single catching cancel boundary event. If the event is found, a post compensation complex gateway is created in order to delay the activation of the outgoing sequence flow from the cancel boundary event until all performed compensatable tasks inside the transaction are compensated. Then, for each compensatable task the handleTaskCompletion and handleCompensation methods are invoked.

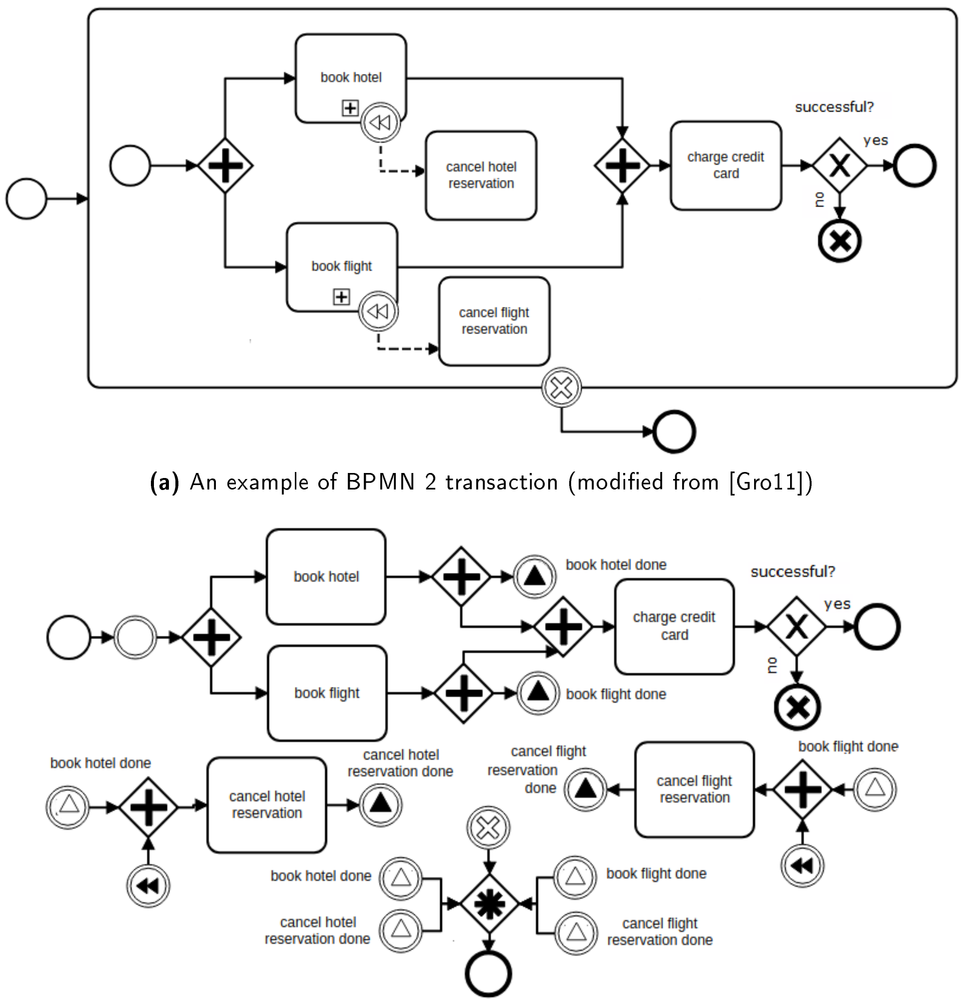
The handleTaskCompletion method creates a send signal event and places it after the given compensatable task (using a newly created gateway to connect it to the other elements if it is needed). Additionally, it creates a receive signal event to catch the generated signal event and adds it to the receivers attribute of the send signal event.

The handleCompensation method starts by finding the receive signal event, which indicates the completion of the given compensatable task. Then, it finds the compensatable tasks that are immediate successors of the current compensatable task within sequences of compensatable tasks and creates the signal events described in the third step.

Figure 5.1.1b demonstrates the result of applying the transaction refinement algorithm on a sample transaction shown in Figure 5.1.1a.

Listing 5.1: Refinement of transactions

```
refine(BPMN2Process proc) {
  foreach (Transaction tran in proc.objects.filter(e | e.isTypeOf('Transaction'))) {

    Event[] cancels = tran.objects.filter(e | e.isTypeOf('CatchingCancelEvent'));
    assert(cancels.length == 1);

    Gateway postCompensation = new ComplexGateway();
    postCompensation.nextFlowObjects = cancels[0].nextFlowObjects;
    cancels[0].nextFlowObjects = {postCompensation};

    foreach(Task start : tran.objects.filter(e | e.isTypeOf('Task') ∧ e.compensation != null) and tran.previousFlowObjects().length == 0) {

        // Allow post compensation flow only when all performed compensatable tasks are compensated
        Event taskDone = new CatchingSignalEvent();
        getDoneSignal(task).receivers.add(taskDone);
        taskDone.nextFlowObjects = {postCompensation};

        Event compensationDone = new CatchingSignalEvent();
        getDoneSignal(task.compensation).receivers.add(compensationDone);
        compensationDone.nextFlowObjects = {postCompensation};
    }

    foreach (CompensatableTask task in tran.objects.filter(e | e.isTypeOf('Task') ∧ e.compensation != null)) {
        handleTaskCompletion(task);
        handleCompensation(cancels[0], task);
    }
  }
}
```



(a) An example of BPMN 2 transaction (modified from [Gro11])

(b) Refined transaction

Figure 5.1.1: BPMN 2 model of Figure 5.1.1a after performing the transaction refinement

Listing 5.2: Refinement of transactions (dealing with task completion)

```
handleTaskCompletion(CompensatableTask task) {
  // A send signal event to indicate the task is done
  Event doneSendEvent = new SendSignalEvent();
  // A receive signal event to catch the signal above
  Event doneReceiveEvent = new CatchingSignalEvent();
  doneSendEvent.receivers = {doneReceiveEvent};
  // Placing the signal event after the task
  if (task.nextFlowObjects == null) {
    task.nextFlowObjects = {doneSendEvent};
  } else {
    Gateway gateway = new InclusiveGateway();
    gateway.nextFlowObjects = task.nextFlowObjects;
    gateway.nextFlowObjects.add(doneSendEvent);
    task.nextFlowObjects = {gateway};
  }
}
```

5.2 Atlas Transformation Language

We have implemented the BPMN 2 to Reo transformation in ATL (ATLAS Transformation Language), which is developed as a part of the ATLAS Model Management Architecture (AMMA) platform [BJT05]. ATL is a hybrid language, meaning that it supports both declarative and imperative programming styles.

A program in ATL consists of several rules that match against the source model elements and generate target elements. Rules in ATL are of three types: matched and lazy rules that are declarative, called rules, which are imperative.

The matched rules define matching conditions for generating target elements out of the source elements and the way to initialize them from the matched source model element. A matched rule contains two mandatory sections, which are the matching and generation patterns; and two optional parts that are local variables definitions and an imperative section.

Local variables are defined by the keyword using. The scope of a local variable is its enclosing rule. The source pattern of a matched rule is defined using the from keyword. By defining an expression on the matching pattern, it is possible to restrict the matching of the source elements to those of choice. A source model element of an ATL transformation can only be matched by one matched rule.

The optional imperative section is defined by the keyword do. The generation part of the rule is specified by the to keyword. Unlike matched rules, a lazy rule is

Listing 5.3: Refinement of transactions (dealing with compensations)

```
handleCompensation(CatchingCancelEvent cancel, CompensatableTask task) {
  Event receiver = getDoneSignal(task).receivers[0];
  CompensatableTask[] nexts = getNextCompensatables(task);
  if (nexts.length == 0) {
    Gateway gateway = new InclusiveGateway();
    cancel.nextFlowObjects.add(gateway);
    receiver.nextFlowObjects = {gateway};
    gateway.nextFlowObjects.add(task.compensation);
  } else {
    // A complex gateway that fires if either all or
    // only the first two of its inputs have flow
    Gateway order = new ComplexGateway();
    cancel.nextFlowObjects.add(order);
    receiver.nextFlowObjects.add(order);

    foreach(CompensatableTask next in nexts) {
      // Event associated with the next compensatable task
      getDoneSignal(next).nextFlowObjects.add(order);

      // Event associated with compensation of the next
      // compensatable task
      Event compensationDone = getDoneSignal(next.compensation).receivers[0];
      getDoneSignal(compensationDone).nextFlowObjects.add(order);
    }
    order.nextFlowObjects.add(receiver);
  }
}
```

Listing 5.4: Definition mapping rule

```
rule mapDefinition {
  from
    def : BPMN2!Definitions
  to
    mod : Reo!Module(
        name <- def.name,
        connectors <- def.rootElements->select(e | e.oclIsKindOf(
            ↪ BPMN2!Process))
        )
}
```

only fired when it is called through another rule.

Imperative programming in ATL is feasible using called rules. They can accept parameters. In order to run a called rule, they need to be explicitly called from an imperative code section.

ATL allows developers to define auxiliary methods, called helpers, which can be called from different parts of the program. An ATL helper consists of a name, a context type, a return type, an ATL expression defining the logic of the helper, and an optional set of parameters defined as pairs of *parameter name* and *parameter type*.

5.3 Mapping BPMN 2 to Reo

We express the mapping in terms of the BPMN 2 and Reo meta-models. Meta-models provide a precise and systematic way to describe valid models.

The conversion begins by matching the BPMN 2 top most element, which according to the BPMN 2 meta-model is Definition. Definition is a container for other BPMN 2 elements.

Similarly, a module serves as the top most container for Reo elements. Both definition and module can be seen as logical elements that are added in the meta-models in order to preserve the process structure. Neither of them exists in the conceptual definition of the notations.

Listing 5.5: Process mapping rule

```
helper context BPMN2!SubProcess def : expanded : Boolean =
  self.flowElements.size() > 0;

helper context BPMN2!FlowNode def : expandedSubProcess : Boolean =
  if not self.oclIsKindOf(BPMN2!SubProcess)
  then false
  else self.expanded
  endif;

rule mapProcess {
  from
    proc : BPMN2!Process
  to
    conn : Reo!Connector(
      name <- proc.name,
      nodes <- proc.flowElements->select(e | e.oclIsTypeOf(BPMN2!
          ↪ Activity) or e.oclIsTypeOf(BPMN2!Event) or e.oclIsTypeOf
          ↪ (BPMN2!Gateway)),
      primitives <- proc.flowElements->select(e | e.oclIsTypeOf(BPMN2
          ↪ !SequenceFlow) or (e.oclIsKindOf(BPMN2!SubProcess) and
          ↪ not e.expanded())),
      subConnectors <- proc.flowElements->select(e | e.
          ↪ expandedSubProcess())
    )
}
```

5.3.1 Definition

We map a definition to a Reo module. The rule in Listing 5.4 carries out this mapping. Similar to all of our mapping rules, it respects the nesting of elements, meaning that the result of mapping an enclosed element is assigned to the mapped parent element. The rule creates a Reo module for the BPMN 2 definition and triggers rules matching the nested processes. The result of the triggered rules will be assigned to connectors inside the created module.

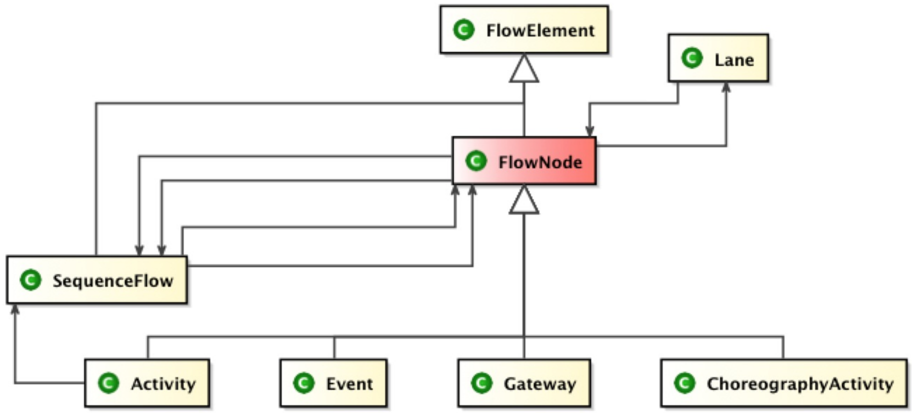


Figure 5.3.1: The FlowNode and its related entities in BPMN 2 EMF meta-model

The select command in the rule collects the processes from the list of elements nested within the rootElements attribute of the definition. RootElement is an abstract type with Process as one of its subtypes. The select command applied on rootElement guarantees that not any other subtype but process will go through this assignment.

The function oclIsKindOf returns *true*, if it is invoked from either an instance of the passed type or an instance of one of its subtypes. Similarly, the function oclIsTypeOf returns *true*, if the element to which it is applied is an instance of the passed type.

5.3.2 Process

We map a BPMN 2 process to a Reo connector in Listing 5.5. Besides creating a connector, the rule initiates the set of nodes, primitives, and subconnectors from the result of mapping the activity, gateway, and event elements, sequenceFlows, and subprocesses, respectively.

When a mapping rule maps an BPMN 2 elements to a mixture of Reo nodes and primitives those types that are the rules in Listing 5.5 does assign to the corresponding attribute in the Reo connector need to be manually assigned to their target attribute of the connector. This is done in the do section of those rules, where we place the recently created primitives inside the corresponding Reo connector. Otherwise, these primitives would be floating inside the model.

We assume that a subprocess is collapsed when it has no inner element. The helper expanded returns *true*, when it is applied on a subprocess with at least one

Listing 5.6: Mapping tasks and collapsed subprocesses

```
rule mapTaskAndCollapsedSubprocess {
  from
    nod : BPMN2!FlowNode(nod.oclIsKindOf(BPMN2!Task) or (nod.
        ↪ oclIsKindOf(BPMN2!SubProcess) and not nod.
        ↪ expandedSubProcess()))
  to
    ndc : Reo!Node,
    fif : Reo!FIFO(sourceEnds <- src, sinkEnds <- snk),
    src : Reo!SourceEnd(node <- ndc),
    snk : Reo!SinkEnd(node <- ndk),
    ndk : Reo!Node
  do {
    ndc.connector.primitives.add(fif);
  }
}
```

inner element. The helper expandedSubProcess serves the same purpose, but with a difference that it is applicable on any flowNode.

As Figure 5.3.1 demonstrates FlowNode mentioned in the rule is the super type of activity, gateway, and event types in the BPMN 2 meta-model.

5.3.3 Task and subprocess

Since a BPMN 2 task represents one unit of work in a process, we map it to a FIFO_1 channel while preserving its incoming and outgoing sequence flows.

Similarly, a collapsed subprocess represents a single step in a process by abstracting away from its inner structure, it resembles a Reo FIFO_1 channel. Listing 5.8 describes the mapping rule for a simple activity and a collapsed subprocess.

Unlike a collapsed subprocess, an expanded subprocess reveals its inner structure. Therefore, we map an expanded subprocess to a Reo subconnector that contains Reo elements mapped from the inner elements of the source subprocess.

The rule in Listing 5.7 first creates a Reo connector, then invokes other rules to map its inner elements, and assigns the result to the generated connector.

Listing 5.7: Mapping an expanded subprocess

```
rule mapExpandedSubprocess {
  from
    subp : BPMN2!SubProcess(subp.expandedSubProcess())
  to
    conn : Reo!Connector(
      name <- subp.name,
      nodes <- subp.flowElements->select(e | e.oclIsTypeOf(BPMN2!
          ↪ Task) or e.oclIsTypeOf(BPMN2!Event) or e.oclIsTypeOf(
          ↪ BPMN2!Gateway)),
      primitives <- subp.flowElements->select(e | e.oclIsTypeOf(
          ↪ BPMN2!SequenceFlow) or (e.oclIsKindOf(BPMN2!SubProcess)
          ↪  and not e.expandedSubProcess())),
      connector <- subp.flowElements->select(e | e.
          ↪ expandedSubProcess()
    )
}
```

5.3.4 Throw and catch events

A catch event catches a trigger from a throw event with the same event type. The type of an event is defined in the eventDefinitions attribute of the event. As mentioned in Chapter 2, event triggers are resolved in one of the following mechanisms:

- *Publication*: message and signal events,
- *Propagation*: escalation and error events,
- *Direct Resolution*: conditional event,
- *Cancellation*: cancel event,
- *Compensation*: compensation event.

We use FIFO channels to queue the event triggers emitted from throw events to be processed by corresponding catch events. This is similar to the approach proposed in [AKM08b] for mapping messages. While the FIFO channels are empty, the throw event can emit a trigger and control flow proceeds to the next step. Meanwhile, the catch event can consume the trigger from the queue asynchronously.

Listing 5.8: Mapping tasks and collapsed subprocesses

```
rule mapTaskAndCollapsedSubprocess {
  from
    nod : BPMN2!FlowNode(nod.oclIsKindOf(BPMN2!Task) or (nod.
        ↪ oclIsKindOf(BPMN2!SubProcess) and not nod.
        ↪ expandedSubProcess()))
  to
    ndc : Reo!Node,
    fif : Reo!FIFO(sourceEnds <- src, sinkEnds <- snk),
    src : Reo!SourceEnd(node <- ndc),
    snk : Reo!SinkEnd(node <- ndk),
    ndk : Reo!Node
  do {
    ndc.connector.primitives.add(fif);
  }
}
```

A limitation of this approach is that when the FIFO is full, the catch event is blocked. To deal with this issue, a lossySync channel can be used to lose the new event triggers if the previously generated events are still waiting to be processed.

When the maximum number of possible event triggers can be calculated, for instance, when the catch event is not reachable from any loop or it is reachable from loops with predefined repeating number, it is possible to use a FIFO_n (which is a sequence of n FIFO_1 channels), where n is the maximum number of loop repetitions.

Listing 5.9 shows the mapping rule for catch events. It creates a Reo node for the source catch event. The name of the generated node is used in Listing 5.10 and 5.11 to connect the catch event to the corresponding throw event using the resolveTemp operator.

Listing 5.10 maps published throw events. The using section finds the corresponding catch events. The to section connects the throw event to its corresponding catch events using FIFO_1 channels. Similarly, Listing 5.11 presents the mapping for propagated throw events. The difference between the two using sections of these rules is due to the difference in trigger forwarding for published and propagated events in BPMN 2. As mentioned in Chapter 2, a propagated trigger is forwarded from its origin to the innermost enclosing level that has an attached catching event that matches the trigger, while propagated event triggers can be caught by any catching event that matches the trigger within any scope where it is published.

Listing 5.9: Mapping non-conditional catch event

```
rule mapCatchingEvent {
  from
    cev : BPMN2!CatchingEvent(cev.eventDefinitions->select(e | tev.
        ↪ eventDefinitions.size() < 2 and not e.oclIsTypeOf(BPMN2!
        ↪ ConditionalEventDefinition)))
  to
    cme : Reo!Node(name <- cev.name)
}
```

The function refImmediateComposite is a special function in ATL, which returns the immediate container. We use it to narrow the scope of search for catch events for the propagated events.

The conditional is directly resolved. This means that there is no throw event for conditional event type, and that such catch events are activated when the corresponding conditions are met.

The rule in Listing 5.12 maps a conditional event to a Reo writer with ability to make infinite I/O request (indicated by assigning *-1* to the writer's request attribute), two nodes that are used to connect the other elements, and a filter channel whose expression attribute matches the source model conditional event.

5.3.5 Gateway

The behavior of a parallel gateway is determined by the number of its incoming and outgoing sequence flows. If it has only one incoming sequence flow, it acts similar to a Reo replicate node. If the number of incoming sequence flows is more that one, the behavior of the gateway is as of a Reo join node as it merges the data items from all the incoming sequence flows and writes the result on the outgoing sequences flows.

The rule in Listing 5.13 generates a Reo node for the matched parallel gateway, wherein the number of incoming sequence flows of the gateway determines the type of the generated Reo node.

Listing 5.10: Mapping published throw message event

```
rule mapPublishedThrowingEvent {
  from
    mte : BPMN2!ThrowingEvent(mte.eventDefinitions->select(e | e.
        ↪ oclIsTypeOf(BPMN2!MessageEventDefinition) or e.oclIsTypeOf
        ↪ (BPMN2!SignalEventDefinition)).size() = 1)
  using {
    cas: Sequence(BPMN2!CatchingEvent) = BPMN2!CatchingEvent.
        ↪ allInstances()->select(e | e.eventDefinitions->first().
        ↪ messageRef = mte.eventDefinitions->first().messageRef or e
        ↪ .eventDefinitions->first().signalRef = mte.
        ↪ eventDefinitions->first().signalRef)->asSequence();
  }
  to
    nod : Reo!Node(name <- mte.name),
    sc1 : Reo!SourceEnd(node <- nod),
    sk1 : Reo!SourceEnd(node <- thisModule.resolveTemp(cat, 'cme')),
    fif : Reo!FIFO(sourceEnds <- sc1, sinkEnds <- sk1)
  do {
    nod.connector.primitives.add(fif);
    for (cat in cas) {
        thisModule.connectByLossyFifo(nod, thisModule.resolveTemp(cat
            ↪ , 'cme'));
    }
  }
}
```

Listing 5.11: Mapping propagated throw events

```
rule mapPropagatedThrowingEvent {
    from
      tev : BPMN2!ThrowingEvent(tev.eventDefinitions->select(e | e.
          ↪ oclIsTypeOf(BPMN2!EscalationEventDefinition) or e.
          ↪ oclIsTypeOf(BPMN2!ErrorEventDefinition)).size() = 1)
    using {
      cas : Sequence(BPMN2!CatchingEvent) = e.refImmediateComposite()
          ↪ .flowElements->select((e | e.eventDefinitions->first().
          ↪ escalationRef=tev.eventDefinitions->first().
          ↪ escalationRef) or (e | e.eventDefinitions->first().
          ↪ errorRef=tev.eventDefinitions->first().errorRef))
    }
    to
       nod : Reo!Node(name <- tev.name)
    do {
         for (cat in cas) {
            thisModule.connectByLossyFifo(nod, thisModule.resolveTemp(
                ↪ cat, 'cme'));
        }
    }
}

rule connectByLossyFifo(nd1 : reo!Node, nd2 : reo!Node) {
   to
      los : Reo!LossySync(sourceEnds <- sc1, sinkEnds <- sk1),
      sc1 : Reo!SourceEnd(node <- nd1),
      sk1 : Reo!SinkEnd(node <- nd3),
      nd3 : Reo!Node,
      fif : Reo!FIFO(sourceEnds <- src, sinkEnds <- snk),
      sc2 : Reo!SourceEnd(node <- nd3),
      sk2 : Reo!SinkEnd(node <- nd2)
   do {
      nd1.connector.nodes.add(nd3);
      nd1.connector.primitives.add(fif);
      nd1.connector.primitives.add(los);
   }
 }
```

Listing 5.12: Mapping conditional event

```
rule mapConditionalEvent {
  from
    cde : BPMN2!CatchingEvent(cde.eventDefinitions->select(e | e.
        ↪ oclIsTypeOf(BPMN2!ConditionalEventDefinition)).size() > 0)
  using {
    cnd : cde.eventDefinitions->select(e | e.oclIsTypeOf(BPMN2!
        ↪ ConditionalEventDefinition).first().condition
  }
  to
      nd1 : Reo!Node,
      nd2 : Reo!Node,
      wrt : Reo!Writer(sinkEnds <- sk1, requests <- -1),
      sk1 : Reo!SinkEnd(node <- nd1),
      sc1 : Reo!SourceEnd(node <- nd1),
      sk2 : Reo!SinkEnd(node <- nd2),
      fil : Reo!Filter(sourceEnds <- sc1, sinkEnds <- sk2, expression
          ↪  <- cnd),
  do {
      nd1.connector.primitives.add(fil);
  }
}
```

Listing 5.13: Mapping parallel gateway

```
rule mapParallelGateway {
   form
       gwy : BPMN2!ParallelGateway
   to
       nod : Reo!Node(
                       name <- gwy.name,
                       type <- if gwy.incoming.size()>0
                               then #JOIN
                               else #REPLICATOR
                               endif
                    )
}
```

Listing 5.14: Mapping inclusive gateway

```
rule mapInclusiveGateway {
   form
      gwy : BPMN2!InclusiveGateway
   to
      nod : Reo!Node(name <- gwy.name)
}

rule mapSequenceFlowOutOfInclusiveGateway {
   from
     seq : BPMN2!SequenceEdge(seq.sourceRef.oclTypeOf(BPMN2!
         ↪ InclusiveGateway))
   to
     fil : Reo!Filter(sourceEnds <- sce, sinkEnds <- ske, expressions
         ↪  <- seq.sourceRef.condition),
     sce : Reo!SourceEnd(node <- seq.sourceRef),
     ske : Reo!SinkEnd(node <- seq.targetRef)
}
```

A diverging inclusive gateway directs the incoming sequence flow to its outgoing sequences, whose conditions are evaluated to *true*. We can achieve the same behavior using a replicate node whose sink ends are connected to filter channels. Each filter channel and its expression corresponds to one of the outgoing sequence flows of the gateway. If the condition is met, then the filter channel passes the incoming data item through. Otherwise, the channel loses the data item. Listing 5.14 shows the rules that carry out the mapping of the inclusive gateway and its outgoing sequence flows.

A diverging exclusive gateway creates alternative paths, where only one path can be taken. Similar to an inclusive gateway, we map an exclusive gateway using a Reo router node and a filter channel for each outgoing sequence flow. Listing 5.15 presents the rule for mapping an exclusive gateway and its outgoing sequence flows.

5.3.6 Transaction

In Listings 5.1, 5.2, and 5.3, we have presented an algorithm to refine BPMN 2 transactions, which introduces two kinds of complex gateways.

1. The compensation order complex gateway that ensures that an activity is

Listing 5.15: Mapping exclusive gateway

```
rule mapExclusiveGateway {
   form
      gwy : BPMN2!ExclusiveGateway
   to
      nod : Reo!Node(name <- gwy.name, type <- #ROUTE)
}

rule mapSequenceFlowOutOfExclusiveGateway {
   from
     seq : BPMN2!SequenceEdge(seq.sourceRef.oclIsTypeOf(BPMN2!
         ↪ ExclusiveGateway))
   to
     fil : Reo!Filter(sourceEnds <- src, sinkEnds <- snk, expressions
         ↪  <- seq.sourceRef.condition),
     src : Reo!SourceEnd(node <- seq.sourceRef),
     snk : Reo!SinkEnd(node <- seq.targetRef)
}
```

only compensated if a cancel event has occurred and the activity has been executed, and in case that there is an activity that needs to be compensated before this activity, it has been compensated.

2. The post compensation complex gateway, which prevents that the outgoing sequence flow of the cancel boundary event is taken before all compensation tasks within the given transaction are completed.

For simplicity, we assume that the transaction refinement step provides a list of the generated complex gateways. Here, we use *orderComplexGateways* and *postComplexGateway* to represent these complex gateways. Alternatively, we could detect them programmatically based on their context in terms of their adjacent elements.

Listing 5.16 presents the rule for mapping a compensation order complex gateway. In this rule and the followings, we capitalize some labels to make it easier to find them later in the figures and to track their usage cross rules. The helper connectingNode defined in Listing 5.17 is used in the mapping of incoming sequence flows to compensation order complex gateway to connect each incoming sequence to its corresponding node that is generated from the complex gateway. Listing 5.18 demonstrates mappings for the sequence flows of the complex gateway.

To make these rules easier to be understood, Figure 5.3.2 illustrates the result of

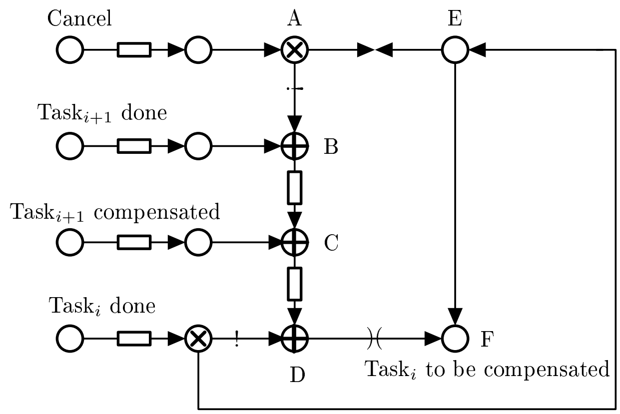


Figure 5.3.2: Mapping of the compensation order complex gateway

applying them to control the flow for compensating the compensatable Task$_i$ with the following compensatable Task$_{i+1}$.

Listing 5.19 shows the rule, which maps the post compensation complex gateway to a join node in Reo. The complex gateway incoming sequence from the catching cancel event is presented in Listing 5.20. Listings 5.21 and 5.22, presents rules, which map the gateway incoming sequence flows from the events signalling the task compensation and the task completion, respectively. Due to lengthiness of these rules, in Figure 5.3.3, we visualize the result of applying them on a transaction with two compensatable tasks: $Task_i$ and $Task_j$ that are in parallel path without any other compensatable tasks ahead of them in a sequence.

5.3.7 Other elements

In general, we map sequence flows to sync channels that coordinate the mapped elements. We map the rest of BPMN 2 flow nodes that are not mapped by the aforementioned rules to Reo nodes.

Since ATL does not provide a mechanism to provide priority over the rules, the

Listing 5.16: Mapping the generated compensation order complex gateway

```
rule mapCompensationOrderComplexGateway {
   from
      cxg : BPMN2:ComplexGateway(thisModule.orderComplexGateways->
         ↪ includes(cxg))
   to
      A : Reo!Node(type <- #ROUTE),
      pab : Reo!PrioritySync(sourceEnds <- sca, sinkEnds <- skb),
      sca : Reo!SourceEnd(node <- A),
      skb : Reo!SinkEnd(node <- B),
      B : Reo!Node(type <- #JOIN),
      fbc : Reo!FIFO(sourceEnds <- scb, sinkEnds <- skc),
      scb : Reo!SourceEnd(node <- B),
      skc : Reo!SinkEnd(node <- C),
      C : Reo!Node(type <- #JOIN),
      fcd : Reo!FIFO(sourceEnds <- scc, sinkEnds <- skd),
      scc : Reo!SourceEnd(node <- C),
      skd : Reo!SinkEnd(node <- D),
      D : Reo!Node(type <- #JOIN),
      sae : Reo!SyncDrain(sourceEnds <- Sequence{sra, sre}),
      sra : Reo!SourceEnd(node <- A),
      sre : Reo!SourceEnd(node <- E),
      E : Reo!Node,
      sef : Reo!Sync(sourceEnds <- sce, sinkEnds <- skf),
      sce : Reo!SourceEnd(node <- E),
      skf : Reo!SinkEnd(node <- F),
      F : Reo!SinkEnd(node <- F),
      pdf : Reo!Sync(sourceEnds <- srd, sinkEnds <- snf),
      srd : Reo!SourceEnd(node <- D),
      snf : Reo!SinkEnd(node <- F)
      do {
         for (e in Sequence{pab, fbc, fcd, pdf, sae) {
            A.connector.primitives.add(e);
         }
      }
}
```

Listing 5.17: Finding the connecting node to a complex gateway

```
helper context BPMN2!FlowNode def : connectingNode(gw :
    ↪ ComplexGateway) : String =
   if self.oclTypeOf(BPMN2!CatchingCancelEvent)
   then 'A'
   else if self.oclTypeOf(BPMN2!CatchingSignalEvent)
        then if thisModule.compensatables.get(gw)->includes(self)
             then 'B'
             else if thisModule.nextCompensatables.get(gw)->includes(
                 ↪ self)
                  then 'C'
                  else if thisModule.nextCompensations.get(gw)->
                      ↪ includes(self)
                        then 'D'
                      endif
                  endif
             endif
        else 'UNKNOWN'
   endif;
```

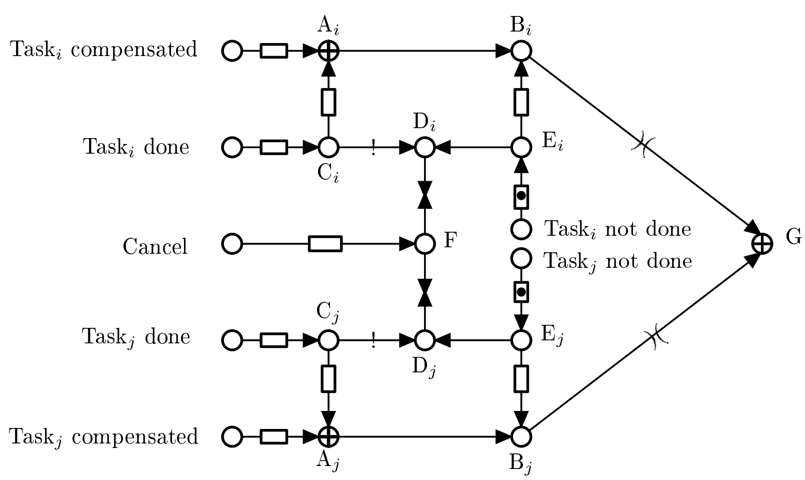


Figure 5.3.3: Mapping of the post compensation complex gateway

rule for mapping the non-specific elements need to have a condition to assure that they do not match any of the existing rules. This is simply achieved by negating the disjunction of the related rules.

Listing 5.18: Mapping incoming flows of the compensation order gateway

```
rule mapSequenceFlowFromCompensatableToOrderComplexGateway {
   from
      seq : BPMN2!SequenceFlow(thisModule.orderComplexGateways->
          ↪ includes(seq.targetRef) and thisModule.nextCompensations
          ↪ .get(gw)->includes(seq.sourceRef))
   to
      fia : Reo!FIFO(sourceEnds <- sca, sinkEnds <- ska),
      sca : Reo!SourceEnd(node <- seq.sourceRef),
      ska : Reo!SinkEnd(node <- thisModule.resolveTemp(seq.targetRef,
          ↪  seq.sourceRef.connectingNode(seq.targetRef))),
      blk : Reo!BlockSync(sourceEnds <- scb, sinkEnds <- skb),
      scb : Reo!SourceEnd(node <- seq.sourceRef),
      skb : Reo!SinkEnd(node <- thisModule.resolveTemp(seq.targetRef,
          ↪  'E'))
}

rule mapSequenceFlowToOrderComplexGateway {
   from
      seq : BPMN2!SequenceFlow(thisModule.orderComplexGateways->
          ↪ includes(seq.targetRef) and not thisModule.
          ↪ nextCompensations.get(gw)->includes(seq.sourceRef))
   to
      fia : Reo!FIFO(sourceEnds <- sca, sinkEnds <- ska),
      sca : Reo!SourceEnd(node <- seq.sourceRef),
      ska : Reo!SinkEnd(node <- thisModule.resolveTemp(seq.targetRef,
          ↪  seq.sourceRef.connectingNode(seq.targetRef)))
}

rule mapSequenceFlowFromOrderComplexGateway {
   from
      seq : BPMN2!SequenceFlow(thisModule.orderComplexGateways->
          ↪ includes(seq.sourceRef))
   to refined
      syn : Reo!Sync(sourceEnds <- src, sinkEnds <- snk),
      src : Reo!SourceEnd(node <- thisModule.resolveTemp(seq.
          ↪ sourceRef, 'F')),
      snk : Reo!SinkEnd(node <- seq.targetRef)
}
```

Listing 5.19: Mapping the post compensation complex gateway

```
rule mapPostCompensationComplexGateway {
   from
      cxg : BPMN2:ComplexGateway(cxg = thisModule.postComplexGateway)
   to
      G : Reo!Node(type <- #JOIN)
}
```

Listing 5.20: Mapping the cancel flow to the post compensation gateway

```
rule mapCancelToPostCompensationComplexGatewaySequenceFlow {
   from
      seq : BPMN2!SequenceFlow(seq.sourceRef.oclTypeOf(BPMN2!
          ↪ CatchingCancelEvent) and seq.targetRef = thisModule.
          ↪ postComplexGateway)
   to
      fia : Reo!FIFO(sourceEnds <- sca, sinkEnds <- ska),
      sca : Reo!SourceEnd(node <- seq.sourceRef),
      ska : Reo!SinkEnd(node <- F),
      F : Reo!Node
}
```

Listing 5.21: Mapping the compensation completion

```
rule mapCompensationToPostCompensationGatewaySequenceFlow {
   from
      seq : BPMN2!SequenceFlow(seq.targetRef = thisModule.
          ↪ postComplexGateway and seq.sourceRef.oclIsKindOf(BPMN!
          ↪ CatchingSignalEvent) and thisModule.nextCompensations.
          ↪ get(seq.targetRef)->includes(seq.sourceRef))
   to
      fi1 : Reo!FIFO(sourceEnds <- sc1, sinkEnds <- sk1),
      sc1 : Reo!SourceEnd(node <- seq.sourceRef),
      sk1 : Reo!SinkEnd(node <- A),
      A : Reo!Node(type <- #JOIN),
      fi2 : Reo!FIFO(sourceEnds <- sc2, sinkEnds <- sk2),
      sc2 : Reo!SourceEnd(node <- thisModule.resolveTemp(seq.
          ↪ sourceRef, 'C')),
      sk2 : Reo!SinkEnd(node <- A),
      sab : Reo!Sync(sourceEnds <- sca, sinkEnds <- skb),
      sca : Reo!SourceEnd(node <- A),
      skb : Reo!SinkEnd(node <- B),
      B : Reo!Node,
      fi3 : Reo!FIFO(sourceEnds <- sce, sinkEnds <- snb),
      sce : Reo!SourceEnd(node <- thisModule.resolveTemp(seq.
          ↪ sourceRef, 'E')),
      snb : Reo!SinkEnd(node <- B),
      bbg : Reo!BlockSync(sourceEnds <- scb, sinkEnds <- skg),
      scb : Reo!SourceEnd(node <- B),
      skg : Reo!SinkEnd(node <- thisModule.resolveTemp(seq.sourceRef,
          ↪  'G'))
      do {
           fil.connector.nodes.add(A);
           fil.connector.nodes.add(B);
      }
}
```

Listing 5.22: Mapping the task completion

```
rule mapCompensatableToPostCompensationGatewaySequenceFlow {
   from
      seq : BPMN2!SequenceFlow(seq.targetRef = thisModule.
          ↪ postComplexGateway and
      seq.sourceRef.oclIsKindOf(BPMN!CatchingSignalEvent) and
            thisModule.nextCompensatables.get(seq.targetRef)->
                ↪ includes(seq.sourceRef))
   to
      fic : Reo!FIFO(sourceEnds <- scf, sinkEnds <- skc),
      scf : Reo!SourceEnd(node <- seq.sourceRef),
      skc : Reo!SinkEnd(node <- C),
      C : Reo!Node,
      pri : Reo!PrioritySync(sourceEnds <- scc, sinkEnds <- skd),
      scc : Reo!SourceEnd(node <- ndc),
      skd : Reo!SinkEnd(node <- D),
      D : Reo!Node,
      sdr : Reo!SyncDrain(sourceEnds <- Sequence{scf, scd}),
      scf : Reo!SourceEnd(node <- thisModule.resolveTemp(seq.
          ↪ targetRef, 'F')),
      scd : Reo!SourceEnd(node <- D),
      syn : Reo!Sync(sourceEnds <- sec, sinkEnds <- snd),
      snd : Reo!SinkEnd(node <- D),
      sec : Reo!SourceEnd(node <- E),
      E : Reo!Node,
      ffe : Reo!FIFO(sourceEnds <- sen, sinkEnds <- ske, full <- true
          ↪ ),
      sen : Reo!SourceEnd(node <- ndt),
      ske : Reo!SinkEnd(node <- D),
      ndt : Reo!Node
      do {
         for (e in Sequence{C, D, E, ndt}) {
            fic.connector.nodes.add(e);
         }
      }
}
```

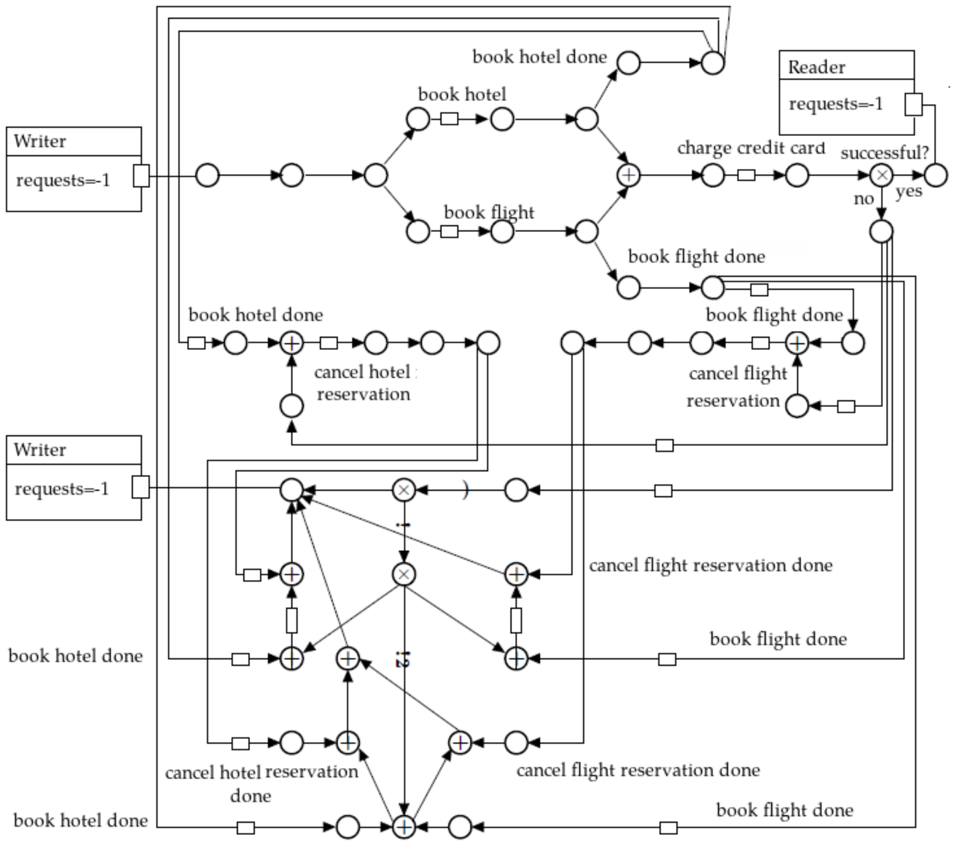


Figure 5.3.4: Mapping the refined BPMN 2 example of Figure 5.1.1b to Reo

5.4 Example

Figure 5.3.4 shows the result of applying the presented BPMN 2 to Reo transformation rules on the refined BPMN 2 model of Figure 5.1.1b.

5.5 Related Work

Several works on the topic of formal semantics of business processes propose a mapping from BPMN to Petri nets [vdA98] e.g. [TSJ10], [DDO08], [DW11], and [MBL+18]. Petri nets constitute a graph-based modeling language for describing distributed systems. Similar to BPMN, Petri nets have a graphical syntax and its execution semantics have exact mathematical definitions.

The obtained Petri nets model can be analyzed using Petri nets analyzing tools such as ProM [vDdMV+05], Yasper [SOP+06], Woflan [VvdAK04], Snoopy[HHL+12], and CPN Tools [JKW07]. Each of these tools performs particular types of analyses. Some tools can only analyze a subset of Petri nets.

Groote et al. in [GMR+06] propose converting the obtained Petri nets models to the process specification language mCRL2 to open up the possibility of automatic verification by the mCRL2 tool-set.

Alternatively, BPMN has been mapped to other formalisms. Wong et al. [WG08] propose a mapping from BPMN to Communicating Sequential Processes (CSP) [Hoa85], a type of process algebra.

Christiansen et al. [CCH11] use a token-based semantics to define formal semantics for BPMN processes. Authors of [ESB14] propose a formal semantics for BPMN processes in Maude [CM02], a logical declarative language based on rewriting logic. Prandi et al. [PQZ08] suggest a translation of BPMN into the process algebra COWS [LPT07].

Braghetto et al. in [BFV11] propose a mapping of BPMN processes into Stochastic Automata Network (SAN) [PA91] - a compositionally built stochastic model. Authors of [MSY14] present a formal model for BPMN processes in terms of Labelled Transition Systems, which are obtained from process algebra encoding. Poizat et al. in [PS12] propose a model transformation into the LOTOS NT process algebra [GLS17].

A drawback of using aforementioned formalisms compared to Petri nets is that they do not preserve the structure of the original BPMN model, as they are lower level languages and at finer granularity compared to BPMN. Reo has graphical syntax and exact mathematical definitions of its execution semantics. It defines a form of coordination in terms of synchronizing, buffering, retaining data, etc., along with constraining its input and output data items. Reo allows hierarchical modeling where arbitrarily complex models can be formed out of simpler ones.

The semantics of Reo is compositional. This means that complex networks can be built by connecting simpler networks. Once a business model is transformed to a Reo network, its behavior can be formally studied using various programs within the Extensible Coordination Tools (ECT) [AKM+08a], a set of Eclipse plug-ins that constitute an integrated development environment for the Reo coordination language.

ECT contains tools for the design [AKM+08a], animation [Kra11], simulation [Kan10], testing [AAA+09], stochastic analysis [ACMM07], verification [KB09, KKdV10, MSA04], execution [Pro11, AJ15, AKM+08a, JSS+12], and model transformation

[CKA10, MSTV07, KMLA11] for Reo networks.

6

A Constraint-Based Semantics Framework for Reo

6.1 Introduction

In Chapter 5, we presented our approach for automatic transformation of business process models into Reo [CKA10]. This enables the use of Reo analysis methods and tools on these processes that originally were not expressed in Reo. Performing analysis on a Reo connector requires the behavior of the connector expressed in one of the formal semantics of Reo.

Each of these formal semantics comes with a set of definitions and operators, which enable calculating semantics of a Reo connector. The straight-forward algorithms of supporting tools for automating this process are developed based on these definitions. These custom algorithms are computationally expensive and not optimized. As a result, in practice the size of a connector they can support is small.

Another inherent limitation of these algorithms stem from that they model data explicitly. As a consequence, in practice the set of input data needs to be limited to a predefined small set. This holds even for connecters with no data-sensitive components, which shows the same behavior for each data item.

Even though different formal semantics of a Reo connector describe the behavior of the same model, since each of them focuses on some behavioral aspects such as context-sensitivity or data-awareness, and ignores some other aspects, it is possible that one aspects of its semantics describes some behavior that another semantics considers invalid. A classical example of this case is when a *lossySync* channel is connected to a $FIFO_1$ channel. The constraint automata and the coloring semantics for this example describe different behavior.

In this chapter, we present a constraint-based framework to derive formal semantics of a Reo connector. We form a constraint by encoding the behavior of constructs of the connector.

Our framework eliminates the result of expressiveness gap among Reo formal semantics by incorporating more than one semantics in deriving the behavior of a Reo connector. This way, we transform problem of calculating formal semantics of a Reo connector into a constraint satisfaction problem, for which efficient and optimized methods and tools exist. We use the symbolic approach to deal with data, i.e, rather than dealing with concrete values, we split the data domain to ranges for which the connector exhibits different behavior.

This work is a necessary step for providing fully automated model checking for data-aware and context-dependent Reo connectors. It can be seen as a generalization of the constraint-based framework presented in [Pro11], that is used as a base for Reo's distributed execution engine. However, there are major differences between them. For instance, the framework for the Reo execution engine only provide support for synchrony and context-sensitivity, while our method deals with priority and data-constraints as well.

6.2 Reo constraint satisfaction problem (RCSP)

In this section, we extend the constraint-based framework in [Pro11] to incorporate all behavioral dimensions addressed by various semantic models for Reo. In our framework, we denote each of these elements by variables over their proper domains.

We relate these variables to each other and restrict possible values they can assume using constraints whose solutions give the underlying formal semantics of the network. In this section, we deal only with connectors whose semantics can be expressed in CASM or CC. Later, we extend our framework to also support priority.

Let $\mathcal{N} = \mathcal{N}^{src} \cup \mathcal{N}^{mix} \cup \mathcal{N}^{snk}$ be the global set of nodes, $\mathcal{M}$ the global set of state memory variables, and $\mathcal{D}$ the global set of numerical data values. The set of primitive ends $\mathcal{P}$ consists of all primitive ends p derived from $\mathcal{N}$ by marking its elements with superscripts c and k, according to the following grammar:

$$p ::= r^c \mid s^k$$

where $r \in \mathcal{N}^{src} \cup \mathcal{N}^{mix}$ and $s \in \mathcal{N}^{snk} \cup \mathcal{N}^{mix}$. Observe that the primitive ends n^c and n^k connect on the common node n.

Let $p \in \mathcal{P}$, $n \in \mathcal{N}$ and $m \in \mathcal{M}$ be a primitive end, a node, and a state memory variable, respectively. A free variable v that occurs in the constraints encoding the behavior of a Reo network has one of the following forms:

- $\tilde{n}$ ranges over $\{\top, \bot\}$ to show presence or absence of flow on the node n.
- $\hat{n}$ ranges over $\mathcal{D}$ to represent the data value passing through the node n.
- $\mathring{m}, \mathring{m}'$ range over $\{\top, \bot\}$ to denote whether or not the state memory variable m is defined in, respectively, the source and the target states of the transition to which the encoded guard belongs.
- $\hat{m}, \hat{m}'$ range over $\mathcal{D}$ to represent the values of the state memory variable m in, respectively, the source and the target states of the transition to which the encoded guard belongs.
- $p^{\triangleright}$ ranges over $\{\top, \bot\}$ to state that the reason for lack of data-flow through the primitive end p originates from the primitive to which p belongs or the context (of this primitive).

Note that not all of the introduced variables are required for encoding the behavior of every Reo network. In presence of context-dependent primitives like *lossySync* or in priority-sensitive networks, constraints include variables of the form $p^{\triangleright}$. For the stateful elements such as FIFO_1, variables like $\mathring{m}, \mathring{m}', \hat{m}$, and $\hat{m}'$ appear in the constraints.

Observe that the interpretation of some of the mentioned variables depends on the values of other variables. Referring to the variable $p^{\triangleright}$ makes sense only if $\tilde{n} = \bot$, where $p = n^c$ or $p = n^k$ (i.e., the primitive end p belongs to the node n); and $\hat{n}$, $\hat{m}$ and $\hat{m}'$ make sense only if $\tilde{n} = \top$, $\mathring{m} = \top$ and $\mathring{m}' = \top$, respectively.

The grammar for a constraint Ψ encoding the behavior of a Reo network is as follows:

$$
\begin{array}{lcll}
t & ::= & \hat{n} \mid \hat{m} \mid \hat{m}' \mid d \mid t \circledast\ d & \text{(terms)} \\
a & ::= & \tilde{n} \mid p^{\triangleright} \mid \mathring{m} \mid \mathring{m}' \mid t = t \mid t < t & \text{(atoms)} \\
\boldsymbol{\psi} & ::= & \top \mid a \mid \neg\boldsymbol{\psi} \mid \boldsymbol{\psi} \wedge\ \boldsymbol{\psi} & \text{(formulae)}
\end{array}
$$

where $d \in \mathcal{D}$ is a constant, $\circledast \in \{+, -, *, /, \%, \hat{}\ \}$, and p is either of the form n^c or n^k.

A solution to a formula $\boldsymbol{\psi}$ is defined over the variable sets $V \times V_d$, where the variables in V are mapped to a value in $\{\bot, \top\}$ and values in V_d are mapped to subsets of D. The satisfaction rules for a solution $\langle \delta, \delta_d \rangle$ are defined as follows:

$$
\begin{array}{ll}
\langle \delta, \delta_d \rangle \models \top & \text{always} \\
\langle \delta, \delta_d \rangle \models \tilde{n} & \text{iff } \delta(\tilde{n}) = \top \\
\langle \delta, \delta_d \rangle \models p^{\triangleright} & \text{iff } \delta(p^{\triangleright}) = \top \\
\langle \delta, \delta_d \rangle \models \mathring{m} & \text{iff } \delta(\mathring{m}) = \top \\
\langle \delta, \delta_d \rangle \models \mathring{m}' & \text{iff } \delta(\mathring{m}') = \top \\
\langle \delta, \delta_d \rangle \models P(t_1, t_2, ..., t_n) & \text{iff } (\delta_d(t_1), \delta_d(t_2), ..., \delta_d(t_n)) \subseteq I(P(t_1, t_2, ..., t_n)) \\
\langle \delta, \delta_d \rangle \models \boldsymbol{\psi}_1 \wedge \boldsymbol{\psi}_2 & \text{iff } \langle \delta, \delta_d \rangle \models \boldsymbol{\psi}_1 \wedge \langle \delta, \delta_d \rangle \models \boldsymbol{\psi}_2 \\
\langle \delta, \delta_d \rangle \models \neg\boldsymbol{\psi} & \text{iff } \langle \delta, \delta_d \rangle \not\models \boldsymbol{\psi}
\end{array}
$$

There exists an associated interpretation, $I(P) \subseteq 2^{D^n}$, for each n-ary predicate P.

Definition 6.2.1 (Reo constraint satisfaction problem) *A Reo constraint satisfaction problem (RCSP) is a tuple $\langle \mathcal{P}, \mathcal{M}, M_0, \mathcal{V}, C \rangle$, where:*

- *$\mathcal{P}$ is a finite set of primitive 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 Reo network.*
- *$\mathcal{V}$ is a set of variables v defined by the grammar*

  $$v ::= \tilde{n} \mid p^{\triangleright} \mid \mathring{m} \mid \mathring{m}' \mid \hat{n} \mid \hat{m} \mid \hat{m}'$$

  for $n \in \mathcal{N}, p \in \mathcal{P}$, and $m \in \mathcal{M}$. The values that the variables of the forms $\hat{n}, \hat{m}$, and $\hat{m}'$ can assume are subsets of $\mathcal{D}$, and the other variables are Boolean, with values in $\{\top, \bot\}$.

- $C = \{C_1, C_2, ..., C_m\}$ *is a finite set of constraints, where each* C_i *is a constraint given by the grammar* Ψ *involving a subset of variables* $V_i \subseteq \mathcal{V}$.

Example 6.2.1 *The RCSP of a* sync *channel with the source end* a *and the sink end* b *is* $\langle \{a, b\}, \emptyset, \emptyset, \{\tilde{a}, \tilde{b}, \hat{a}, \hat{b}\}, \tilde{a} \Leftrightarrow \tilde{b} \wedge \tilde{a} \Rightarrow (\hat{a} = \hat{b})\rangle$. *The solutions for this constraint problem give the behavior of the* sync *channel as the channel allows data-flow on its source end iff its sink end can dispense it simultaneously (which agrees with the semantics of this channel as defined in other formal models of Reo). In case of data-flow, the values of the data items passing through the ends of this channel are equal.*

We obtain the constraints corresponding to a Reo network by composing the RCSPs of its constituents as defined below.

Definition 6.2.2 (Composition) *The composition of two RCSPs* $\rho_1 = \langle \mathcal{P}_1, \mathcal{M}_1, M_{0,1}, \mathcal{V}_1, C_1\rangle$ *and* $\rho_2 = \langle \mathcal{P}_2, \mathcal{M}_2, M_{0,2}, \mathcal{V}_2, C_2\rangle$ *is defined as follows:*

$$\rho_1 \odot \rho_2 = \langle \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{M}_1 \cup \mathcal{M}_2, M_{0,1} \cup M_{0,2}, \mathcal{V}_1 \cup \mathcal{V}_2, C_1 \wedge C_1\rangle$$

However, connecting two Reo networks must not introduce incorrect data-flow possibilities. This is done by enforcing a restriction on the possible solutions through the following axiom:

Axiom 6.2.1 (Mixed node axiom) *When two Reo networks connect on the common node* x, *where* x^c *is a source end in one network and* x^k *is a sink end in the other, the following constraint must hold:*

$$\neg\tilde{x} \Leftrightarrow (x^{c\triangleright} \vee x^{k^{\triangleright}})$$

The *mixed node axiom*, which applies to all mixed nodes in a network, states that a node x cannot produce the reason for no-flow all by itself.

6.2.1 Encoding Reo elements in RCSPs

Table 6.2.2 summarizes the constraint encodings associated with commonly used Reo elements. If a Reo network does not contain any context-dependent channel, the variables encoding the context-dependency can be ignored in its RCSP. Table 6.2.1 shows the encoding of Reo elements from Table 6.2.2 where the context variables are removed. Note that in these tables, a and b denote the source and the sink ends of a primitive, respectively, and that *dom* refers to the domain of the given function

Table 6.2.1: Context-independent encoding of Reo primitives

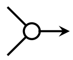
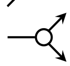
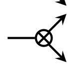
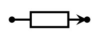
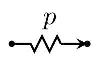
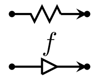
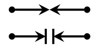
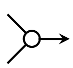
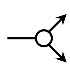
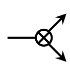
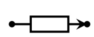
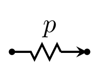
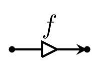
Channel	Constraints
	$\psi_{Sync}(a,b) : \tilde{a} \Leftrightarrow \tilde{b} \wedge \tilde{a} \Rightarrow (\hat{a} = \hat{b})$
	$\psi_{SyncDrain}(a_1,a_2) : \tilde{a}_1 \Leftrightarrow \tilde{a}_2$
	$\psi_{AsyncDrain}(a_1,a_2) : \neg(\tilde{a}_1 \wedge \tilde{a}_2)$
	$\psi_{LossySync} : \tilde{b} \Rightarrow \tilde{a} \wedge \tilde{b} \Rightarrow (\hat{a} = \hat{b})$
	$\psi_{Merger}(a_{0..i},b) : \tilde{b} \Leftrightarrow (\bigvee_i \tilde{a}_i) \bigwedge_{j,j\neq i} \neg(\tilde{a}_i \wedge \tilde{a}_j) \wedge \tilde{a}_i \Rightarrow (\hat{a}_i = \hat{b})$
	$\psi_{Replicator}(a,b_{0..i}) : \tilde{a} \Leftrightarrow (\bigwedge_i \tilde{b}_i) \wedge \tilde{a} \Rightarrow (\bigwedge_i(\hat{b}_i = \hat{a}))$
	$\psi_{Router}(a,b_{0..i}) : \tilde{a} \Leftrightarrow (\bigvee_i \tilde{b}_i) \bigwedge_{j,j\neq i} \neg(\tilde{b}_i \wedge \tilde{b}_j) \wedge \tilde{b}_i \Rightarrow (\hat{b}_i = \hat{a})$
	$\psi_{FIFO_1}(a,b,m) : \tilde{a} \Rightarrow (\neg\mathring{m} \wedge \mathring{m}' \wedge (\hat{m}' = \tilde{a})) \wedge \tilde{b} \Rightarrow (\mathring{m} \wedge \neg\mathring{m}' \wedge (\hat{m} = \tilde{b})) \wedge (\neg\tilde{a} \wedge \neg\tilde{b}) \Rightarrow (\mathring{m} \Leftrightarrow \mathring{m}' \wedge \mathring{m} \Rightarrow (\hat{m} = \hat{m}))$
	$\psi_{Filter}(a,b,P) = \tilde{b} \Rightarrow (\tilde{a} \wedge \hat{b} \in dom(P) \wedge P(\hat{a}) \wedge (\hat{a} = \hat{b}))$
	$\psi_{Transformer}(a,b,f) = \tilde{b} \Rightarrow (\tilde{a} \wedge \hat{b} \in dom(f)) \wedge \tilde{b} \Rightarrow (\hat{b} = f(\hat{a}))$

Table 6.2.2: Context-dependent encoding of Reo primitives

Channel	Constraints
	$\psi_{Sync}(a,b) : \tilde{a} \Leftrightarrow \tilde{b} \wedge \tilde{a} \Rightarrow (\hat{a} = \hat{b}) \wedge \neg(a^{c\triangleright} \wedge b^{k\triangleright})$
	$\psi_{SyncDrain}(a_1,a_2) : \tilde{a}_1 \Leftrightarrow \tilde{a}_2 \wedge \neg(a_1^{c\triangleright} \wedge a_2^{c\triangleright})$
	$\psi_{AsyncDrain}(a_1,a_2) : \tilde{a}_1 \Rightarrow (\neg\tilde{a}_2 \wedge a_2^{c\triangleright}) \wedge \tilde{a}_2 \Rightarrow (\neg\tilde{a}_1 \wedge a_1^{c\triangleright})$
	$\psi_{LossySync}(a,b) : \tilde{b} \Rightarrow \tilde{a} \wedge \tilde{b} \Rightarrow (\hat{a} = \hat{b}) \wedge \neg a^{c\triangleright} \wedge \neg\tilde{a} \Rightarrow b^{k\triangleright}$
	$\psi_{Merger}(a_{0..i},b) : \tilde{a}_i \Leftrightarrow \tilde{b} \wedge \tilde{a}_i \Rightarrow (\hat{a}_i = \hat{b}) \wedge \neg\tilde{b} \Rightarrow ((\neg b^{k\triangleright} \bigwedge_i a_i^{c\triangleright}) \vee (b^{k\triangleright} \wedge \neg a_i^{c\triangleright} \bigwedge_{j,j!=i} a_j^{k\triangleright}))$
	$\psi_{Replicator}(a,b_{0..i}) : \tilde{a} \Leftrightarrow \bigwedge_i \tilde{b}_i \wedge (\tilde{a} \Rightarrow \bigwedge_i(\hat{b}_i = \hat{a})) \wedge \neg\tilde{a} \Rightarrow ((\neg a^{c\triangleright} \bigwedge_i b_i^{k\triangleright}) \vee (\neg b_i^{k\triangleright} \bigwedge_{j,j\neq i} b_j^{k\triangleright} \wedge a^{c\triangleright}))$
	$\psi_{Router}(a,b_{0..i}) : \tilde{a} \Leftrightarrow (\bigvee_i \tilde{b}_i) \bigwedge_{j,j\neq i} \neg(\tilde{b}_i \wedge \tilde{b}_j) \wedge \tilde{b}_i \Rightarrow (\hat{b}_i = \hat{a}) \wedge \tilde{a} \Leftrightarrow (\neg a^{c\triangleright} \vee \neg(\bigvee_i b_i^{k\triangleright}))$
	$\psi_{FIFO_1}(a,b,m) : \tilde{a} \Rightarrow (\neg\mathring{m} \wedge \mathring{m}' \wedge (\hat{m}' = \hat{a})) \wedge \tilde{b} \Rightarrow (\mathring{m} \wedge \neg\mathring{m}' \wedge (\hat{m} = \hat{b})) \wedge (\neg\tilde{a} \wedge \neg\tilde{b}) \Rightarrow (\mathring{m} \Leftrightarrow \mathring{m}' \wedge \mathring{m} \Rightarrow (\hat{m} = \hat{m}')) \wedge \neg\mathring{m} \Rightarrow b^{k\triangleright} \wedge \mathring{m} \Rightarrow a^{c\triangleright}$
	$\psi_{Filter}(a,b,P) = \tilde{b} \Rightarrow (\tilde{a} \wedge \hat{a} \in dom(P) \wedge P(\hat{a})) \wedge \tilde{b} \Rightarrow (\hat{a} = \hat{b}) \wedge (\neg\tilde{a} \Rightarrow (\neg a^{c\triangleright} \Leftrightarrow b^{k\triangleright})) \wedge (\tilde{a} \wedge \neg\tilde{b} \Rightarrow b^{k\triangleright})$
	$\psi_{Transformer}(a,b,f) = \tilde{b} \Rightarrow (\tilde{a} \wedge \hat{a} \in dom(f)) \wedge \tilde{b} \Rightarrow (\hat{b} = f(\hat{a})) \wedge \neg(a^{c\triangleright} \wedge b^{k\triangleright})$

or predicate. In the case of elements with more than one source or sink ends, we use indices.

The intuition behind these constraints is that their solutions reflect the semantic model of each element as given by CASM and CC.

Example 6.2.2 *Figure 6.2.1 shows a Reo network that consists of a* transformer *channel with the function* $3 * \hat{a}$*, whose domain is the set of numbers* $Number$ *and a* filter *channel with the condition* $\hat{b}\%2 = 0$ *and domain* $Number$*.*

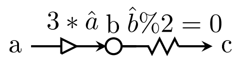


Figure 6.2.1: A data-aware Reo connector

Since none of the Reo primitives in Figure 6.2.1 is context-dependent, we use the constraints corresponding to the primitives in this network as defined in Table 6.2.1.

$$\psi_{Transformer}(a, b, 3 * \hat{a}) = \tilde{a} \Leftrightarrow \tilde{b} \wedge \tilde{a} \Rightarrow (\hat{a} \in Number \wedge \hat{b} = 3 * \hat{a})) \tag{6.1}$$

$$\psi_{Filter}(b, c, \hat{b}\%2 = 0) = \tilde{c} \Rightarrow (\tilde{b} \wedge \hat{b} \in Number \wedge (\hat{b}\%2 = 0)) \tag{6.2}$$

Equation 6.1 states that flow occurs on the source end of the *transformer* channel iff it occurs on its sink end. In addition, flow can exist only if the data item that enters the source end of the channel is a number. In this case, the data item written on the sink end is three times the value of the source data item.

Equation 6.2 expresses that flow on the source end of the *filter* channel leads to flow on its sink end, iff the data item belongs to the channel's accepting pattern (which is $\hat{b}\%2 = 0$).

In this case, the value of data items passing through the ends are equal. No flow through the sink end c is either due to no flow on b or that the incoming data item does not satisfy the accepting pattern. As mentioned, the conjunction of these constraints (subject to Axiom 7.2.1, which trivially holds in this case) encodes the behavior of the given Reo network.

6.2.2 Solving RCSPs

In this section, we show how to obtain the solutions of RCSPs. Since Reo Constraint Satisfaction Problems (RCSPs) have predicates with free variables of types Boolean

($\{\top, \bot\}$) and data ($\mathcal{D}$), a SAT-solver or a numeric constraint solver cannot solve them alone. Satisfiability Modulo Theories (SMT) [BSST09] solvers find solutions for propositional satisfiability problems where propositions are either Boolean or constraints in a specific theory.

However, SMT-solvers are not applicable in our case either, because unlike SAT-solvers they find only an instance of a solution as opposed to the complete set of answers. Another drawback of most SAT- and SMT-solvers is that they work only on quantifier-free formulae, while we use existential quantifies to implement the *hiding* operator of constraint automata (see Section 6.3).

To generate the CASM corresponding to a given Reo network, we need all solutions and thus resort to a hybrid approach that uses both SAT-solvers and Computer Algebra Systems (CASs), namely, REDUCE [Ray87], which is a system for general algebraic computations.

First, we form a pure Boolean constraint system by substituting data dependent constraints with new Boolean variables and find all solutions for the new constraints using a SAT-solver. Then, by substituting each such solution into the original constraints, we obtain a data dependent constraint satisfaction problem that a CAS can solve symbolically. From these solutions, we extract a CASM corresponding to the Reo network encoded by the original set of constraints. Our approach avoids state explosion by treating data constraints symbolically. In the following, we elaborate on our approach.

In an RCSP $\langle \mathcal{P}, \mathcal{M}, M_0, \mathcal{V}, C \rangle$, let $\mathcal{V}_B$ and $\mathcal{V}_D$ be the sets of free Boolean and free data variables of C, respectively, where $\mathcal{V} = \mathcal{V}_B \cup \mathcal{V}_D$, and let A_D be the set of atomic predicates of C containing data variables. The following is our procedure for solving C.

1. We obtain C_B from C by replacing every occurrence of $x \in A_D$ with a unique new Boolean variable $y \notin \mathcal{V}$. For example, for $C = (\tilde{c} \Rightarrow \tilde{b}) \wedge (\tilde{c} \Rightarrow (\hat{b} \in Number \Rightarrow \hat{b}\%2 = 0))$ in Figure 6.2.1, we obtain C_B as $(\tilde{c} \Rightarrow \tilde{b}) \wedge (\tilde{c} \Rightarrow (y_1 \Rightarrow y_2))$ where y_1 and y_2 replace $\hat{b} \in Number$ and $\hat{b}\%2 = 0$, respectively.

2. An off-the-shelf SAT-solver can find the set of solutions S_B for C_B. We define the finite set of constraints $C\left[S_B\right] = \{C\left[v_1, v_2, \ldots, v_n \setminus z_1, z_2, \ldots z_n\right] \mid$ for all distinct $v_i \in \mathcal{V}_B, 1 \leq i \leq n = |\mathcal{V}_B|, z_i \in S\left(v_i\right), S \in S_B\}$.

3. Every $C_D \in C\left[S_B\right]$ is a numerical constraint satisfaction problem, which we (symbolically) solve using a Computer Algebra System. Every solution to each C_D along with the SAT solution $S \in \mathcal{S}_B$ that produced $C_D \in C\left[S_B\right]$ in the previous step, constitute a solution to the RCSP.

Using the presented technique, we obtain the solutions for the RCSP corresponding to Examples 6.2.2 as follows:

1. $\langle\{\tilde{a} = \bot, \tilde{b} = \bot, \tilde{c} = \bot\}, \top\rangle$,
2. $\langle\{\tilde{a} = \top, \tilde{b} = \bot, \tilde{c} = \bot\}, \hat{a} \notin Number\rangle$,
3. $\langle\{\tilde{a} = \top, \tilde{b} = \top, \tilde{c} = \bot\}, \hat{a} \in Number \wedge \hat{b} = 3 * \hat{a} \ \wedge \ \hat{b}\%2 \neq 0\rangle$,
4. $\langle\{\tilde{a} = \top, \tilde{b} = \top, \tilde{c} = \top\}, \hat{a} \in Number \wedge \ \hat{b} = 3 * \hat{a} \ \wedge \ \hat{b}\%2 = 0 \ \wedge \ \hat{b} = \hat{c}\rangle$.

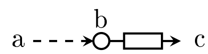


Figure 6.2.2: A context-dependent Reo connector

Example 6.2.3 *Figure 6.2.2 depicts a Reo network that consists of a* lossySync *channel and a* FIFO_1 *channel connecting on the node b.*

Since the Reo network in Figure 6.2.1 contains a *lossySync* that is a context dependent channel, we use the context-aware RCSP encoding from Table 6.2.2:

$$\psi_{LossySync}(a, b) = \tilde{b} \Rightarrow (\tilde{a} \wedge (\hat{a} = \hat{b})) \wedge \neg a^{c^{\triangleright}} \wedge \neg\tilde{a} \Rightarrow b^{k^{\triangleright}}. \tag{6.3}$$

$$\psi_{FIFO_1}(b, c, m) = \tilde{b} \Rightarrow (\neg\mathring{m} \wedge \mathring{m}' \wedge (\hat{m}' = \hat{b})) \wedge \tilde{c} \Rightarrow (\mathring{m} \wedge \neg\mathring{m}' \wedge (\hat{m} = \hat{c})) \wedge (\neg\tilde{b} \wedge \neg\tilde{c}) \Rightarrow ((\mathring{m} \Leftrightarrow \mathring{m}') \wedge \mathring{m} \Rightarrow (\hat{m} = \hat{m}')) \wedge \neg\mathring{m} \Rightarrow c^{c^{\triangleright}} \wedge \mathring{m} \Rightarrow b^{k^{\triangleright}}. \tag{6.4}$$

Equation 6.3 states that flow on the sink end of the *lossySync* is due to flow on its source end. If there is flow on the sink end of the *lossySync*, the data items exchanged at the source and the sink ends are the same. However, it is possible that the source end has flow, but the sink end does not. In this case, the reason for no flow comes from the environment with which the sink end communicates. The third possible behavior of the channel is that there is no flow on the source end due to the environment, in which case the channel provides a reason for no flow on its sink end.

Equation 6.4 expresses the behavior of the $FIFO_1$ channel as follows: The flow on the source end of the channel states that the value of the variable representing the state memory (of the current state) is undefined. The flow on the source end defines the state memory variable for the next state to contain the value of the incoming data item. On the other hand, flow on the sink end means that the value of the state memory variable is defined. The data item leaving the sink end is equivalent to the buffer's data item. In addition, the value of the state memory variable becomes undefined in the next state. If there is no flow on the ends, the variables related to the states stay the same. Being empty, the $FIFO_1$ channel provides a reason for no flow on its sink end, while being full does so on the source end of the channel.

The solutions for the RCSP 6.4, (where for brevity, we omit the values of the variables representing the context, such as $b^{c\triangleright}$) are as follows:

1. $\langle\{\tilde{a} = \bot, \tilde{b} = \bot, \tilde{c} = \bot, \mathring{m} = \bot, \mathring{m}' = \bot\}, \top\rangle$,
2. $\langle\{\tilde{a} = \top, \tilde{b} = \top, \tilde{c} = \bot, \mathring{m} = \bot, \mathring{m}' = \top\}, \hat{a} = \hat{b} \wedge \hat{m}' = \hat{b}\rangle$,
3. $\langle\{\tilde{a} = \top, \tilde{b} = \bot, \tilde{c} = \bot, \mathring{m} = \top, \mathring{m}' = \top\}, \hat{m} = \hat{m}'\rangle$,
4. $\langle\{\tilde{a} = \bot, \tilde{b} = \bot, \tilde{c} = \bot, \mathring{m} = \top, \mathring{m}' = \top\}, \hat{m} = \hat{m}'\rangle$,
5. $\langle\{\tilde{a} = \top, \tilde{b} = \bot, \tilde{c} = \bot, \mathring{m} = \top, \mathring{m}' = \bot\}, \hat{m} = \hat{c}\rangle$,
6. $\langle\{\tilde{a} = \bot, \tilde{b} = \bot, \tilde{c} = \top, \mathring{m} = \top, \mathring{m}' = \bot\}, \hat{m} = \hat{c}\rangle$.

6.2.3 Constructing CASM

In order to construct the CASM from the set of solutions S for an RCSP $\langle\mathcal{P}, \mathcal{M}, M_0, \mathcal{V}, C\rangle$, we first define

- $\mathcal{N} = \{n \mid n^c \in \mathcal{P} \vee n^k \in \mathcal{P}\}$

and then map each solution $\langle s, s_d\rangle \in S$ into a transition $q \xrightarrow{N,g} p$ as follows:

- $q = \langle\{m \mid m \in \mathcal{M}, s\,(\mathring{m}) = \top\}\rangle$,
- $p = \langle\{m \mid m \in \mathcal{M}, s\,(\mathring{m}') = \top\}\rangle$,
- $N = \{n \mid n \in \mathcal{N}, s\,(\tilde{n}) = \top\}$,
- The data constraint g is (a syntactic variant of) s_d.

We obtain the CASM $A = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ from the set $\longrightarrow$ of all transitions generated above, where:

- $Q = \{q \mid q \xrightarrow{N,g} p \vee p \xrightarrow{N,g} q\}$,
- $q_0 = \langle\{m \mid m \in M_0, s(\mathring{m}) = \top\}\rangle$,

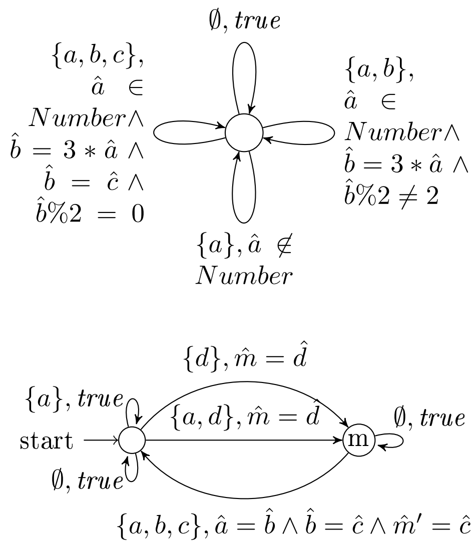


Figure 6.2.3: CASMs generated for Figures 6.2.1 and 6.2.2

- $\mathcal{M}$ is the same $\mathcal{M}$ as in the RCSP.

Applying the above procedure to the solutions of RCSPs constraints generates their corresponding CASMs. For instance, the first solution for the constraints in Example 6.2.2 generates the transition $q \xrightarrow{\emptyset, true} q$, where q is the only state of the CASM, which has no state memory variable. This is so because the set of variables of the form $\mathring{m}$ is empty. Also, the transition has no synchronizing port, because the value of every one of the variables $\tilde{a}, \tilde{b}$ and $\tilde{c}$ is $\perp$. Figures 6.2.3a and 6.2.3b show the CASMs derived from the RCSPs in Examples 6.2.1 and 6.2.2.

Our approach deals with data in a symbolic fashion, where we partition the global set of data values to equivalence classes toward which a Reo network behaves differently. This is in contrast with the traditional way of dealing with data in the formal semantics of Reo (and other models), where they consider a different state for each possible value that can be stored in buffers and a distinct transition for each data value passing through the ports.

Our symbolic approach allows working with an infinite data domain. In addition, rather than implementing the highly time- and memory-demanding custom-made algorithms to generate Reo formal semantics, we use the efficient SAT-solvers and computer algebra systems to solve constraints whose solutions are equivalent to these models.

An experimental study done on the efficiency of using SAT-solvers to generate

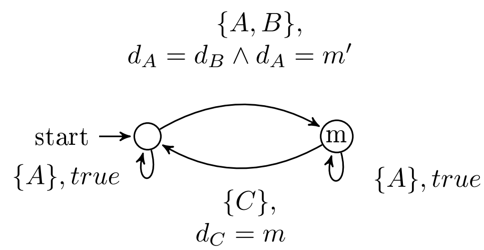


Figure 6.2.4: CASM for Figure 6.2.2

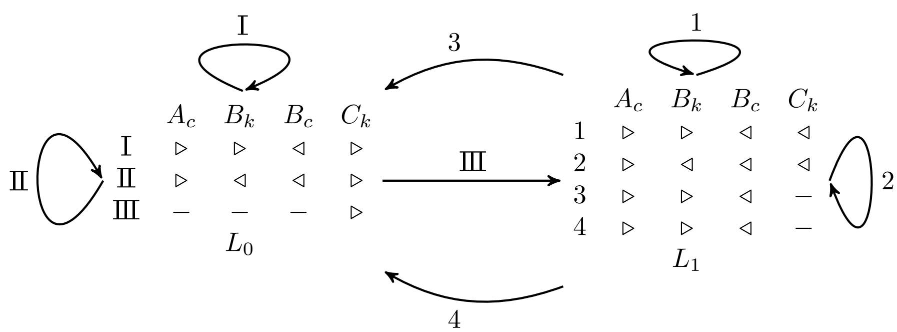


Figure 6.2.5: CC for Figure 6.2.2

Reo formal semantics [Pro11] compares two prototypes based on constraint satisfaction techniques and connector coloring semantics, without taking data constraints in consideration. The results illustrate that the approach based on constraint solving scales better and is more efficient. In chapter 7 we present an evaluation through a case study, which affirms this conclusion.

6.3 Hiding

We use *hiding* to abstract from internal transitions. This is a mechanism to support hierarchy and is used to create components.

The author in [Pro11] proposes applying the existential quantifier to the constraints encoding of the behavior of a network to abstract from internal ports and their corresponding data variables. Similarly, we use existential quantifiers such as $\exists \tilde{e}, \hat{e}, e^{\triangleright} : C$, where C is the RCSP of a Reo network and e is an internal node to hide.

Although several algorithms exist for the problem of quantifier elimination in Boolean algebra and first order logic [BZ07] [Abd02] [Dav88], we are not aware of any working tool that does quantifier elimination on Boolean algebraic formulae. Therefore, our tool implements the *hiding* operator as defined for CASM.

Hiding the internal nodes on some transitions can make the set of their synchronized nodes empty. Here, we refer to such a transition as an *empty* transition, if the free variables of its guard are merely state memory variables. Under some circumstances, we can merge the source and the target states of empty transitions. Let q and p be two states in a CASM such that $q \xrightarrow{\emptyset,g} p$. The following are the conditions under which the state p can merge into the sate q:

1. The states q and p have the same number of state memory variables.

2. The guard g consists of the conjunction of the predicates of the form of $x = y'$, for $x, y \in \mathcal{M}$. This way, g defines a correspondence relation between the state memory variables of the state q and those of the state p.

3. For each transition $q \xrightarrow{N,g'} r$ where $r \notin \{p, q\}$, there is a transition $p \xrightarrow{N,g''} r$ such that $g' \Leftrightarrow g''_g$, where g''_g is obtained from g by replacing all occurrences of the next state memory variable y' with the next state memory variable x', if g contains $x = y'$ for state memory variables $x, y \in \mathcal{M}$.

4. For each transition $r \xrightarrow{N,g'} p$ where $r \notin \{p, q\}$, there is a transition $r \xrightarrow{N,g''} q$ such that $g'' \Leftrightarrow g'_g$, where g'_g is derived from g by substituting all occurrences of the state memory variable x in g with the state memory variable x, if g contains $x = y'$ for state memory variables $x, y \in \mathcal{M}$.

Provided that the above conditions hold, the state p merges into the state q as follows:

1. We eliminate the transition $q \xrightarrow{\emptyset,g} p$.

2. We remove the state p after substituting y, y', and p with x, x', and q in all transitions. Observe that such substitutions convert the non-eliminated transitions between the states q and p into loops over the state q.

Example 6.3.1 *Figure 6.3.1 shows a* FIFO_2 *derived from composing two* FIFO_1*s. The CASM corresponding to the* FIFO_2 *is in Figure 6.3.2a. Figure 6.3.2b depicts the CASM resulting from hiding the mixed node b. Figure 6.3.2c presents the result of eliminating the empty transitions.*

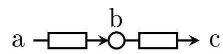
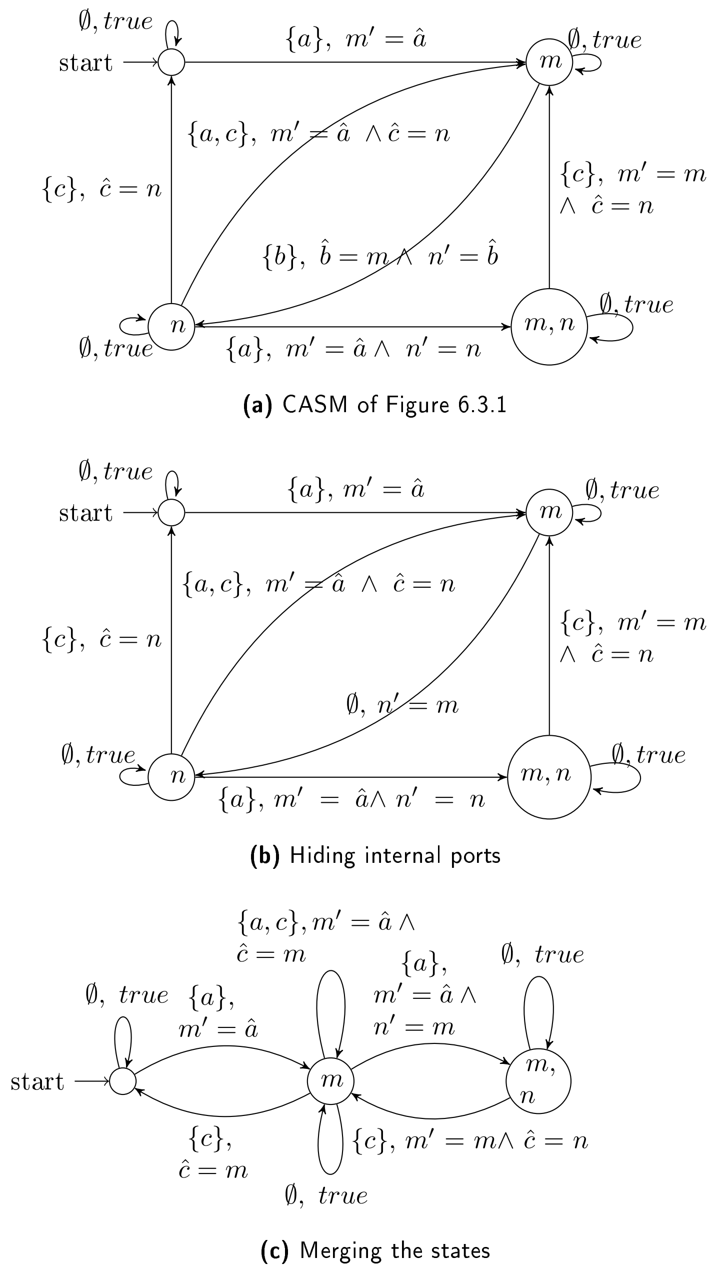


Figure 6.3.1: Two $FIFO_1$s forming $FIFO_2$



(a) CASM of Figure 6.3.1

(b) Hiding internal ports

(c) Merging the states

Figure 6.3.2: Hiding the empty transition and merging its source and target states for the CASM of $FIFO_2$ in Figure 6.3.1

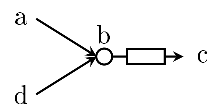


Figure 6.4.1: A sample Reo network

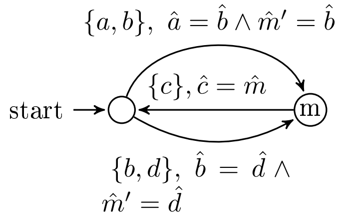


Figure 6.4.2: CASM corresponding to Figure 6.4.1

6.4 Correctness and compositionality

CASM and CC model the presence and the absence of data flow on a Reo network at different levels of granularity. For instance, Figure 6.4.2 and Figure 6.4.3 are the CASM and CC semantics for the Reo network in Figure 6.4.1. As the figures show, the node b in CASM is mapped to three primitive ends in CC, which do not necessarily have the same coloring.

In this section, we formally investigate the relation between the solutions of the RCSP for a given Reo network and its CC and CASM semantics. However, we first need to present some definitions.

For a given network R with $A = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$, its CASM and $C = \langle \mathcal{P}, \mathcal{L}, L_0, \eta \rangle$, its CC, we define the function $O_R : \mathcal{P} \rightarrow \mathcal{N}$ as it maps each CC port to its corresponding node in CASM.

Definition 6.4.1 (Correlation $\sim$) *Let $A = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ be a CASM and $C = \langle \mathcal{P}, \mathcal{L}, L_0, \eta \rangle$ be a CC. We define the relation $\sim : Q \times \mathcal{L}$, as follows:*

- *$q_0 \sim L_0$, if $\mathcal{N} = \bigcup_{p \in \mathcal{P}} O_R(p)$.*
- *For each $p \in Q$ and $L' \in \mathcal{L}$, $p \sim L'$ if the following conditions hold:*

  1. *There exists $q \in Q$ and $L \in \mathcal{L}$ such that $q \xrightarrow{N,g} p$ and $L' = \eta(L, l)$, where $l \subset L$,*
  2. *For all $n \in N, n = O_R(p) \Leftrightarrow l(e) = -$,*
  3. *$q \sim \mathcal{L}$.*

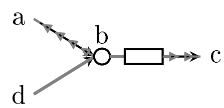


Figure 6.4.3: A coloring annotated state of the CC corresponding to Figure 6.4.1

If a relation $\sim$ exists between Q and $\mathcal{L}$, then we say that A correlates to C, written as $A \sim C$.

It is easy to see that if A and C belong to the same Reo network, then $q_0 \sim L_0$. Therefore, $A \sim C$.

Definition 6.4.2 (*id* mapping) *For the CASM $A = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ and the coloring semantics $C = \langle \mathcal{P}, \mathcal{L}, L_0, \eta \rangle$ such that $A \sim C$, the function $id : \mathcal{L} \rightarrow 2^Q$ correlates coloring tables with subsets of Q such that $id(L)$ returns the set of all $q \in Q$ wherein the data-flow possibilities resulting from the outgoing transitions of q correspond to the data-flow possibilities prescribed by the coloring table L.*

The following example illustrates Definition 6.4.2.

Example 6.4.1 *Figure 6.2.4 and Figure 6.2.5 are, respectively, the CASM and the CC of the Reo network of Figure 6.2.2.*

Note that we have modified the presentation of the CC to resemble the CASM structure. For instance, the transition $L_1 \xrightarrow{i} L_2$ represents $L_2 = \eta$ (L_1, $cols_{L_1}$ $[i]$), where the $cols_{L_1}$ is the possible colorings for each coloring table as shown in the example.
Let q designate the state without a state memory variable in the CASM of Figure 6.2.5, and let p designate the state with the state memory variable m. Then, according to Definition 6.4.1, $q \sim L_0$ and $p \sim L_1$.

Definition 6.4.3 (Memory cells of a state) *We use $\mathcal{M}_q$ to denote the set of all $m \in \mathcal{M}$ that syntactically appear as m in a data constraint g on an outgoing transition $q \xrightarrow{N,g} p$ of the state q. Analogously, we use $\mathcal{M}'_q$ to denote the set of all $m \in \mathcal{M}$ that syntactically appear as m' in a data constraint g on an incoming transition $p \xrightarrow{N,g} q$ of the state q. We call $\mathcal{M}_q$ and $\mathcal{M}'_q$, respectively, the* accessed *and the* updated *memory cells of the state q.*

Definition 6.4.4 (Encoding a Reo network) *For the semantics for a Reo network R as $A = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ and $C = \langle \mathcal{P}, \mathcal{L}, L_0, \eta \rangle$, the RSCP $\Psi = \langle \mathcal{P}, \mathcal{M}, M_0, \mathcal{V}, \mathcal{C} \rangle$ encodes R in terms of its CASM and CC semantics iff the following conditions hold:*

1. *For all solution pairs* $\langle s, s_d \rangle \vDash \Psi$*, there exist a transition* $q \xrightarrow{N,g} p$ *and a colorings* $l \in L \in \mathcal{L}$ *such that*

   (a) *for all* $m \in \mathcal{M}$*,* $m \in \mathcal{M}_q$ *iff* $s(\mathring{m}) = \top$

   (b) *for all* $m \in \mathcal{M}$*,* $m \in \mathcal{M}_p$ *iff* $s(\mathring{m}') = \top$

   (c) *for all* $n \in \mathcal{N}$*,* $n \in N$ *iff* $s(\tilde{n}) = \top$

   (d) *for all* $\hat{v} \in \mathcal{V}$*,* $[g]\, \hat{v} \backslash s_d(\hat{v})$

   (e) *for all* $p \in \mathcal{P}$*,* $s(\tilde{p}) = \top$ *iff* $l(n) = -$ *coloring*

   (f) *for all* $p \in \mathcal{P}$*,* $s(e^{\triangleright}) = \top$ *where* e *is either* p^c *or* p^k *iff* $l(n) = \triangleleft$

   (g) *for all* $p \in \mathcal{P}$*,* $s(e^{\triangleright}) = \bot$ *where* e *is either* p^c *or* p^k *iff* $l(n) = \triangleright$

   (h) *for all* $p \in \mathcal{P}$ *such that* $p^c \cup p^k \subset \mathcal{P}$*, if* $sol(\tilde{p}) = \bot$*, then* $p^{c\triangleright} \vee p^{k^{\triangleright}}$*.*

2. *For all transitions* $q \xrightarrow{N,g} p$*, and colorings* $l \in L \in \mathcal{L}$ *such that* $q \sim L$ *and* $p \sim \eta(L, l)$*, there exists a solution* $\langle s, s_d \rangle$ *such that*

   (a) *for all* $\mathring{m} \in \mathcal{V}$*,* $s(\mathring{m}) = \top$ *iff* $q \in id(L)$ *and* $m \in \mathcal{M}_q$

   (b) *for all* $\mathring{m}' \in \mathcal{V}$*,* $s(\mathring{m}') = \top$ *iff* $p \in id(\eta(L, l))$ *and* $m \in \mathcal{M}'_p$

   (c) *for all* $\tilde{n} \in \mathcal{V}$ *iff* $n \in N$ *and* $l(n) = -$

   (d) *for all* $\hat{v} \in \mathcal{V}$*,* $g\,[v]\,\backslash s_d\,(\hat{v})$

   (e) *for all* $e^{\triangleright} \in \mathcal{V}$*, where* e *is either* n^c *or* n^k*,* $s(e^{\triangleright}) = \top$ *iff* $n \notin N$ *and* $l(n) = \triangleleft$

   (f) *for all* $e^{\triangleright} \in \mathcal{V}$*, where* e *is either* n^c *or* n^k*,* $s(e^{\triangleright}) = \bot$ *iff* $n \notin N$ *and* $l(n) = \triangleright$

The purpose of this encoding is to obtain the behavior of the Reo network as specified in both its CASM and CC semantics by solving the RCSP ψ.

Theorem 6.4.1 (Correctness) *For the CASM* $A = (Q, \mathcal{N}, \rightarrow, q_0, \mathcal{M})$ *and the CC* $C = \langle \mathcal{P}, \mathcal{L}, L_0, \eta \rangle$ *such that* $A \sim C$*, let* Ψ *be the RCSP encoding* A *and* C*. The CASM* $A' = (Q', \mathcal{N}', \rightarrow', q'_0, \mathcal{M}')$ *and the CC* $C' = \langle \mathcal{P}', \mathcal{L}', L'_0, \eta' \rangle$ *extracted from the solutions of* Ψ *are refinements of* A *and* C *and* $A' \sim C'$*.*

Proof For all solution $s \vDash \Psi$, there is a coloring l' and a transition $q' \xrightarrow{N',g'} p'$ such that the first part of Definition 6.4.4 holds:

- $l' \in L$,
- $q' \xrightarrow{N',g'} p'$,

- $q' \in Q$,
- $p' \in Q$.

We construct $A' = (\bigcup (q' \cup p')\,, \mathcal{N}', \rightarrow', q'_0, \mathcal{M}')$ and $C' = \langle \mathcal{P}', \mathcal{L}', L'_0, \eta' \rangle$ from the solutions, where $A' \sqsubseteq A$ and $C' \sqsubseteq C$.

Lemma 6.4.1 *Assume the condition (1) of Definition 6.4.4 holds for two RC-SPs $\Psi_1 = \langle \mathcal{P}_1, \mathcal{M}_1, \mathcal{M}_{0,1}, \mathcal{V}_1, \mathcal{C}_1 \rangle$ and $\Psi_2 = \langle \mathcal{P}_2, \mathcal{M}_2, \mathcal{M}_{0,2}, \mathcal{V}_2, \mathcal{C}_2 \rangle$ for automata $A_1 = (Q_1, \mathcal{N}_1, \rightarrow_1, q_{0_1}, \mathcal{M}_1)$ and $A_2 = (Q_2, \mathcal{N}_2, \rightarrow_2, q_{0_2}, \mathcal{M}_2)$ and colorings $C_1 = \langle \mathcal{P}_1, \mathcal{L}_1, L_{0_1}, \eta_1 \rangle$ and $C_2 = \langle \mathcal{P}_2, \mathcal{L}_2, L_{0_2}, \eta_2 \rangle$. Then the condition (1) of Definition 6.4.4 holds for $Psi_1 \odot Psi_2$, $A_1 \bowtie A_2$ and $C_1 \bullet C_2$.*

Proof Assume $\langle s, s_d \rangle \vDash \Psi_1$ and $\langle s, s_d \rangle \vDash \Psi_2$. Let $\langle s_1, s_{d_1} \rangle$ and $\langle s_2, s_{d_2} \rangle$ be the images of $\langle s, s_d \rangle$ over $\mathcal{V}_1$ and $\mathcal{V}_2$, respectively. Then $\langle s_1, s_{d_1} \rangle \vDash \Psi_1$ and $\langle s_2, s_{d_2} \rangle \vDash \Psi_2$ and for each $v \in \mathcal{V}_1 \cap \mathcal{V}_2$, $s_1(v) = s_2(v)$ and $s_{d_1}(v) = s_{d_2}(v)$.
Therefore, there exist transitions $q_1 \xrightarrow{N_1, g_1}_1 p_1$ and $q_1 \xrightarrow{N_2, g_2}_2 p_2$ and colorings $l_1 \in L_1 \in \mathcal{L}_1$ and $l_2 \in L_2 \in \mathcal{L}_2$ such that the condition (*1*) of Definition 6.4.4 holds.
For each $\tilde{v} \in \mathcal{V}_1 \cap \mathcal{V}_2$, $s_1(\tilde{v}) = \top$ iff $v \in N_1$ and $v \in N_2$. Therefore, $N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1$, which means $\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge q_2} \langle p_1, p_2 \rangle$.
For each $\tilde{v} \in \mathcal{V}_1 \cap \mathcal{V}_2$, $s_1(\tilde{v}) = \top$ iff $l_1(O_R(n)) = -$ and $l_2(O_R(n)) = -$, $s_1(\tilde{v}) = \bot \wedge s_1(v^{\triangleright}) = \top$ iff $l_1(O_R(n)) = \triangleright$ and $l_2(O_R(n)) = \triangleright$, and $s_1(\tilde{v}) = \bot \wedge s_1(v^{\triangleright}) = \bot$ iff $l_1(O_R(n)) = \triangleright$ and $l_2(O_R(n)) = \triangleleft$.
On the other hand,

- for all $m \in \mathcal{M}$, $m \in \mathcal{M}_q$ iff $s(\mathring{m}) = \top$
- for all $m \in \mathcal{M}$, $m \in \mathcal{M}_p$ iff $s(\mathring{m}') = \top$
- for all $n \in \mathcal{N}$, $n \in N$ iff $s(\tilde{n}) = \top$
- for all $\hat{v} \in \mathcal{V}$, $[g]\,\hat{v} \backslash s_d(\hat{v})$
- for all $p \in \mathcal{P}$, $s(\tilde{p}) = \top$ iff $l(n) = -$
- for all $p \in \mathcal{P}$, $s(e^{\triangleright}) = \top$ where e is either p^c or p^k iff $l(n) = \triangleleft$
- for all $p \in \mathcal{P}$, $s(e^{\triangleright}) = \bot$ where e is either p^c or p^k iff $l(n) = \triangleright$

Therefore, the condition (*1*) of Definition 6.4.4 holds for $\Psi_1 \odot \Psi_2$, $A_1 \bowtie A_2$ and $C_1 \bullet C_2$.

Lemma 6.4.2 *Assume the condition (2) of Definition 6.4.4 holds for two RCSPs* $\Psi_1 = \langle \mathcal{P}_1, \mathcal{M}_1, \mathcal{M}_{0,1}, \mathcal{V}_1, \mathcal{C}_1 \rangle$ *and* $\Psi_2 = \langle \mathcal{P}_2, \mathcal{M}_2, \mathcal{M}_{0,2}, \mathcal{V}_2, \mathcal{C}_2 \rangle$ *and for CASMs* A_1 *and* A_2 *and CCs* C_1 *and* C_2*. Then the condition (2) of Definition 6.4.4 holds for* $\Psi_1 \odot \Psi_2$*,* $A_1 \bowtie A_2$ *and* $C_1 \bullet C_2$*.*

Proof Consider the solutions $\langle s_1, s_{d_1} \rangle \models \Psi_1$ and $\langle s_2, s_{d_2} \rangle \models \Psi_2$ such that $\langle s_1, s_{d_1} \rangle$ encodes $q_1 \xrightarrow{N_1, g_1}_1 p_1$ and $l_1 \in C_1$ and $\langle s_2, s_{d_2} \rangle$ encodes $q_2 \xrightarrow{N_2, g_2}_2 p_2$ and $l_2 \in C_2$. Then, $\langle s, s_d \rangle \models \Psi_1 \odot \Psi_2$, where $\langle s, s_d \rangle = \langle s_1 \cup s_2, s_{d_1} \cup s_{d_2} \rangle$. Here, we distinguish between two cases:

- For all $v \in dom(s_1) \cap dom(s_2)$ and for all $\hat{v} \in dom(s_{d_1}) \cap dom(s_{d_2})$, $s_1(v) = s_2(v)$ and $s_{d_1}(\hat{v}) = s_{d_2}(\hat{v})$.
- Otherwise.

The former case describes valid solutions. For two transitions $q_1 \xrightarrow{N_1, g_1}_1 p_1$ and $q_2 \xrightarrow{N_2, g_2}_2 p_2$, we have $\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle$ iff $N_1 \cup \mathcal{N}_2 = N_2 \cup \mathcal{N}_1$.
For two colorings l_1 and l_2, the coloring $l = l_1 \odot l_2$ is valid iff either $e^c \in dom(l_1)$ and $e^k \in dom(l_2)$ and $\neg(l_1(e^c) = \triangleleft \wedge l_2(e^k) = \triangleleft)$ or $e^k \in dom(l_1)$ and $e^c \in dom(l_2)$, $\neg(l_1(e^k) = \triangleleft \wedge l_2(e^c) = \triangleleft)$.
For all $n \in N_1$ and $n \in N_2$, $s_1(n) = \top$, $s_2(n) = \top$, $n \in N_1 \cap N_2$ and $N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1$ means that $\{n | \tilde{n} \in \mathcal{P}_1 \wedge s_1(\tilde{n}) = \top \wedge \tilde{n} \in \mathcal{P}_2\} = \{n | \tilde{n} \in \mathcal{P}_2 \wedge s_2(\tilde{n}) = \top \wedge \tilde{n} \in \mathcal{P}_1\}$. So, $\{n | \tilde{n} \in \mathcal{P}_1 \cup \mathcal{P}_2 \wedge s_1(\tilde{n}) = \top\} = \{n | \tilde{n} \in \mathcal{P}_1 \cap \mathcal{P}_2 \wedge s_2(\tilde{n}) = \top\}$. This means that for all $\tilde{n} \in \mathcal{P}_1 \cap \mathcal{P}_2$, $s_1(\tilde{n}) = s_2(\tilde{n})$.
For all $q_1 \in Q_1$, $m \in M(q_1)$ iff $s_1(m) = \top$ and $q_2 \in Q_2$, $m \in M(q_2)$ iff $s_2(m) = \top$. Since $\mathcal{M}_1 \cap \mathcal{M}_2 = \emptyset$, $M(\langle q_1, q_2 \rangle) = M(q_1) \cup M(q_2)$, $m \in M(\langle q_1, q_2 \rangle)$ iff $s_1(m) = \top \vee s_2(m) = \top$.
If $s_{d_1} \Rightarrow g_1$ and $s_{d_2} \Rightarrow g_2$, then $s_{d_1} \cup s_{d_2} \Rightarrow g_1 \wedge g_2$.
The latter gives invalid solutions, which are impossible. Therefore, the condition (*2*) of Definition 6.4.4 holds for $\Psi_1 \odot \Psi_2$, $A_1 \bowtie A_2$ and $C_1 \bullet C_2$.

Theorem 6.4.2 (Compositionality) *If* Ψ_1 *encodes the automaton* A_1 *and the CC* C_1 *and* Ψ_2 *encodes the automaton* A_2 *and the CC* C_2*, then* $\Psi_1 \odot \Psi_2$ *encodes the automaton* $A_1 \bowtie A_2$ *and the CC* $C_1 \bullet C_2$*.*

Proof It follows directly from Lemmas 6.4.1 and 6.4.2.

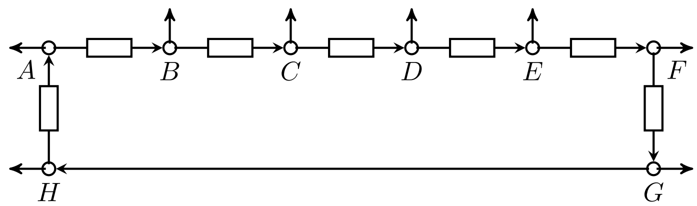


Figure 6.4.4: 7-Sequencer

6.4.1 Performance evaluation

In the remainder of this section, we perform an evaluation on the performance of the presented constraint-based approach along with a brief comparison with the existing approaches, namely, connector coloring and constraint automata.

The execution time of the algorithm depends on the number of states of the given RLTS and the time required to solve the constraints encoding of the network. Thus, to study the performance of our framework and to compare it with the existing approaches in computing operational semantics of Reo networks, we choose the case of *N-Sequencer*, which consists of N *FIFO* channels that are circularly connected.

In this example, adding each $FIFO_1$ channel doubles the number of the states in the corresponding semantics 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 only will influence the size of the constraint. In addition, adding more $FIFO_1$ channels to the network increases both the number of the states and the size of the constraint capturing the semantics of the network. Since we are using 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 are just interested in the approximate growth of the constraints.

Figure 6.4.4 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. The implementation of our approach is in Java 8. We have used Reduce Algebra System[Ray87] to compute the conjunctive normal form of the constraints and to solve them. We have also experimented with an optimization on the number of the variables used in the constraints by substituting equal variables with a single variable. The result of the original and the optimized approaches are presented with red and blue square markers, respectively.

Figure 6.4.5a presents the average time required for computing a single solution of the RCSP of a *N-Sequencer*. Figure 6.4.5b demonstrates the relation between N and the size of the RCSP's constraints of a *N-Sequencer*. This is an indication of complexity of the constraint that needs to be solved. Note that the number of solutions for RCSP of a *N-Sequencer* is *2N*, which equals to the number of transitions in the corresponding RLTS. Finally, Figure 6.4.5c illustrates the total time required to compute all solutions of a RCSP's constraint of a *N-Sequencer*. Figure 6.4.5d shows the time consumed to calculate the coloring semantics and the constraint automata semantics of *N-Sequencer*s using the ECT tool-set. For $N = 16$, the computation of coloring semantics fails with the stack overflow error. The same happens while computing the constraint automata semantics for $N = 21$.

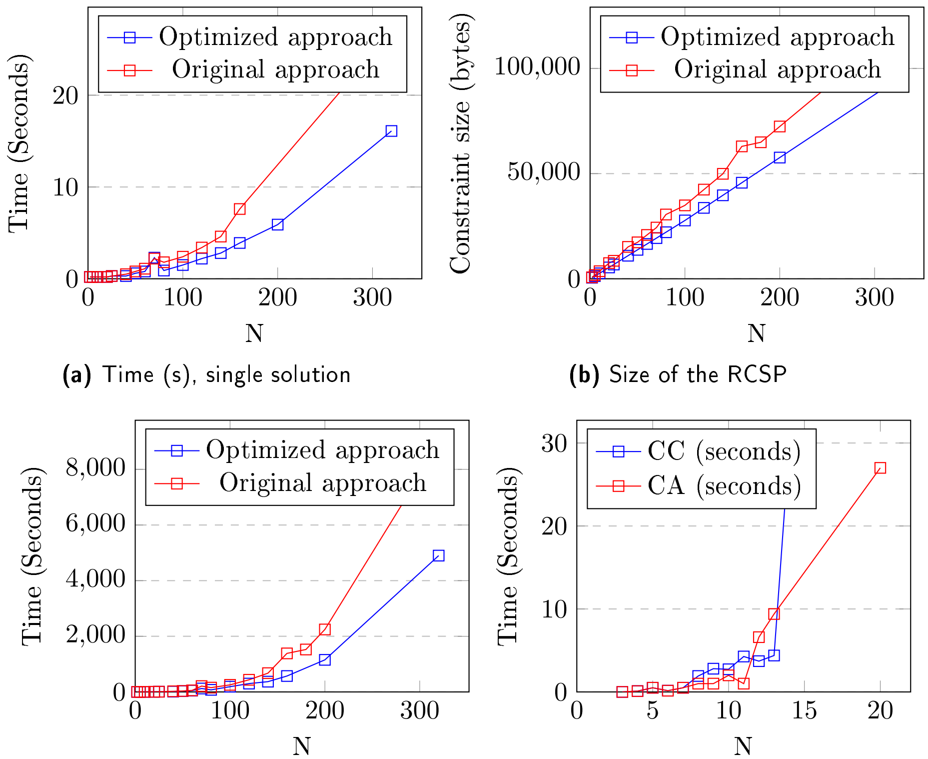
As the results show our approach can handle bigger models compared to the existing ECT tools. It is interesting to observe that the difference between the original and optimized approaches becomes more significant for bigger values of N. Another possible optimization point is the call to Reduce program that is currently implemented by invoking the program externally. We expect a better performance due to reduction of external invocation overhead by including the source code of the Reduce Algebra System in our tool.

6.5 Conclusions

In this chapter, we have presented a constraint-based framework that encodes the semantics of Reo networks as constraint satisfaction problems whose predicates are either Boolean propositions or numerical constraints. We presented a hybrid approach to find the solutions for these problems.

An advantage of our approach is that it treats data constraints symbolically to mitigate the state explosion problem. From this solution, we construct the semantic model corresponding to a Reo network in the form of constraint automata with state memory.

Our framework supports *product* and *hiding* operations on constraint automata.



(a) Time (s), single solution

(b) Size of the RCSP

(c) Time (s), calculating the RLTS

(d) Time (s), building CA and connector coloring table

Figure 6.4.5: Performance evaluation based on N-Sequencer network

We have implemented and integrated our approach as a tool in the ECT. In the next section, we use this framework to encode priority. It makes our work the most expressive framework that exists to analyze Reo networks.

7

Priority

7.1 Introduction

Priority is an important concept in modeling workflows. For instance, modeling compensation and error handling requires a mechanism to express priority of some flow alternatives over others. In the context of Reo, priority can be utilized as a mechanism to impose preferences on the otherwise non-deterministic choices.

Arbab et al. in [ABS15] introduce a compositional approach to model priority and a priority-aware formal semantics for Reo, named Constraint Automata with Priority (CAP), which is an extension of constraint automata.

This approach, which distinguishes between where priority is originated from and where it must be applied i.e. non-deterministic choices, consists of the following elements:

- A primitive to impose priority that is *prioritySync*,

- A mechanism to propagate priority from the location it is imposed through the network,

- A mechanism to block the propagation of priority in desired places using one of the following primitives:
  - *BlockSourceSync*, which stops propagation of priority coming from its source end toward its sink;
  - *BlockSinkSync*, which blocks propagation of priority from its sink end toward its source;
  - *BlockSync* that stops propagation of priority on both ends.
- Means to affect the otherwise non-deterministic choices by priority.

CAP is an expressive formalism for supporting priority in Reo. However, its operations to manipulate CAPs are computationally expensive, if they are implemented in a straight-forward fashion.

The practical needs for dealing with large models of realistic business processes currently complicates direct use of automata-based semantic models. In this chapter, we extend our constraint-based framework presented in Chapter 6 to support priority in Reo. The rest of this chapter is organized as follows: In Section 7.2, we introduce priority flow in Reo along with a constraint-based semantics for it. In Section 7.3, we extend our approach to support numeric priorities. In Section 7.4, we show the application of our constraint-based approach. In Section 7.5, we overview related work. Finally, in Section 7.6, we conclude the chapter and outline future work.

7.2 Priority flow

We distinguish between two types of priority on a node:

- when the node is imposing the priority to be propagated, which we call it *innate* priority,
- when the node has obtained the priority through propagation, we refer to it as *acquired.*

Both ends of *prioritySync* 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*, *syncDrain*) has *acquired* priority, the other end has *innate* priority.

However, in the case of non-synchronous channels (e.g., *FIFO*, *asyncDrain*) and also the priority blocking channels, their ends can only have *acquired* priority. We update the constraint-based framework for Reo presented in Chapter 6 to support priority and the priority propagation mechanism, which we informally described above. In the rest of this chapter, we omit data constraints when defining behavior of Reo elements. Data constraints are irrelevant for priority flow and were thoroughly covered in Chapter 6.

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 $m \in \mathcal{M}$:

- $\tilde{n} \in \{\top, \bot\}$ shows presence or absence of data-flow on n;
- $\mathring{m}, \mathring{m}' \in \{\top, \bot\}$ denotes whether or not the state memory variable m is defined in the source and the target states of the transition, respectively;
- $n^{\triangleright} \in \{\top, \bot\}$ indicates the reason for lack of data-flow on n originating from the primitive or the context (of this primitive), respectively;
- $n^{!^\bullet}$, $n^{!^\circ} \in \{\top, \bot\}$ 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:

$$
\begin{array}{lcll}
a & ::= & \tilde{n} \mid n^{!^\bullet} \mid n^{!^\circ} \mid n^{\triangleright} \mid \mathring{m} \mid \mathring{m}' & \text{(atoms)}, \\
\boldsymbol{\psi} & ::= & \top \mid a \mid \neg\boldsymbol{\psi} \mid \boldsymbol{\psi} \wedge \boldsymbol{\psi} & \text{(formulae)}
\end{array}
$$

A solution to $\boldsymbol{\psi}$ is a map from the variable sets V to a value in $\{\bot, \top\}$. The satisfaction rules for a solution $\langle\delta\rangle$ are satisfaction in propositional logic. We denote the set of all solutions for Ψ as $\mathfrak{S}(\Psi)$.

In Chapter 6 we have introduced RCSP. Here we extend the definition of RCSP and its composition operator with the priority notion and some axioms, which assist in incorporating priority in our constraint-based framework.

Definition 7.2.1 (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*

$$v ::= \tilde{n} \mid n^{\triangleright} \mid \mathring{m} \mid \mathring{m}' \mid n^{!^\circ} \mid n^{!^\bullet} \text{ for } n \in \mathcal{N} \text{ and } m \in \mathcal{M}.$$

- $C = \{C_1,\ C_2,\ ...,\ C_m\}$ *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 7.2.2 (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 7.2.1 (Join axiom) *To propagate no-flow reasons, when a source end* c *and a sink end* k *from two networks, the following holds:*

$$\neg\tilde{c} \Leftrightarrow \neg\tilde{k} \Leftrightarrow (c^{\triangleright} \vee k^{\triangleright}).$$

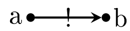
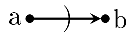
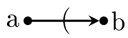
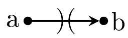
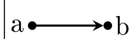
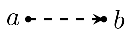
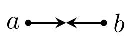
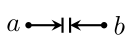
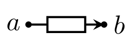
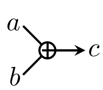
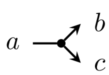
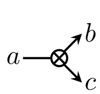
Axiom 7.2.2 (Priority join axiom) *When a source end* c *and a sink end* k *from two networks connect, this holds:*

$$(c^{!^{\circ}} \vee c^{!^{\bullet}} \Leftrightarrow k^{!^{\circ}} \vee k^{!^{\bullet}}) \wedge (c^{!^{\circ}} \wedge k^{!^{\circ}} \Leftrightarrow c^{!^{\bullet}} \vee k^{!^{\bullet}}).$$

Axiom 7.2.3 (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:*

$$(\neg\tilde{x} \wedge (x^{!^{\circ}} \ \vee\ x^{!^{\bullet}}) \wedge \tilde{y} \wedge \neg(y^{!^{\circ}} \ \vee\ y^{!^{\bullet}})) \Rightarrow \neg x^{\triangleright}.$$

Table 7.2.1: Constraint encoding of Reo with priority

Channel	Constraints
	$\psi_{PrioSync}(a,b) : (\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^{\triangleright} \wedge b^{\triangleright}) \wedge a^{!\bullet} \wedge b^{!\bullet}$
	$\psi_{BlkSrcSync}(a,b) : (\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^{\triangleright} \wedge b^{\triangleright}) \wedge \neg b^{!\bullet}$
	$\psi_{BlkSnkSync}(a,b) : (\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^{\triangleright} \wedge b^{\triangleright}) \wedge \neg a^{!\bullet}$
	$\psi_{BlkSync}(a,b) : (\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^{\triangleright} \wedge b^{\triangleright}) \wedge \neg a^{!\bullet} \wedge \neg b^{!\bullet}$
	$\psi_{Sync}(a,b) : (\tilde{a} \Leftrightarrow \tilde{b}) \wedge \neg(a^{\triangleright} \wedge b^{\triangleright}) \wedge ((\neg a^{!\bullet} \wedge \neg a^{!\circ} \wedge \neg b^{!\bullet} \wedge \neg b^{!\circ}) \vee (a^{!\bullet} \wedge \neg b^{!\bullet} \wedge b^{!\circ}) \vee (\neg a^{!\bullet} \wedge a^{!\circ} \wedge b^{!\bullet}))$
	$\psi_{LossySync}(a,b) : \tilde{b} \Rightarrow \tilde{a} \wedge \neg a^{\triangleright} \wedge \neg \tilde{a} \Rightarrow b^{\triangleright} \wedge ((\neg a^{!\bullet} \wedge \neg a^{!\circ} \wedge \neg b^{!\bullet} \wedge \neg b^{!\circ}) \vee (a^{!\bullet} \wedge \neg b^{!\bullet} \wedge b^{!\circ}) \vee (\neg a^{!\bullet} \wedge a^{!\circ} \wedge b^{!\bullet}))$
	$\psi_{SyncDrain}(a_1,a_2) : \tilde{a} \Leftrightarrow \tilde{b} \wedge \neg(a^{\triangleright} \wedge b^{\triangleright}) \wedge ((\neg a^{!\bullet} \wedge \neg a^{!\circ} \wedge \neg b^{!\bullet} \wedge \neg b^{!\circ}) \vee (a^{!\bullet} \wedge \neg b^{!\bullet} \wedge b^{!\circ}) \vee (\neg a^{!\bullet} \wedge a^{!\circ} \wedge b^{!\bullet}))$
	$\psi_{AsyncDrain}(a_1,a_2) : \tilde{a} \Rightarrow (\neg \tilde{b} \wedge b^{\triangleright}) \wedge \tilde{b} \Rightarrow (\neg \tilde{a} \wedge a^{\triangleright}) \wedge \neg a^{!\bullet} \wedge \neg b^{!\bullet}$
	$\psi_{FIFO_1}(a,b,m) : (\tilde{a} \Rightarrow \neg \mathring{m} \wedge \mathring{m}') \wedge (\tilde{b} \Rightarrow \mathring{m} \wedge \neg \mathring{m}') \wedge (\neg \tilde{a} \wedge \neg \tilde{b}) \Rightarrow (\mathring{m} \Leftrightarrow \mathring{m}') \wedge (\neg \mathring{m} \Rightarrow b^{\triangleright}) \wedge (\mathring{m} \Rightarrow a^{\triangleright}) \wedge (\neg a^{!\bullet} \wedge \neg b^{!\bullet})$
	$\psi_{Merger}(a,b,c) : (\tilde{a} \vee \tilde{b}) \Rightarrow \tilde{c} \wedge \neg(\tilde{a} \wedge \tilde{b}) \wedge \neg \tilde{c} \Rightarrow ((\neg c^{\triangleright} \wedge a^{\triangleright}) \vee (c^{\triangleright} \wedge \neg a^{\triangleright} \wedge b^{\triangleright}) \vee (c^{\triangleright} \wedge \neg b^{\triangleright} \wedge a^{\triangleright})) \wedge b^{\triangleright})) \wedge (c^{!\circ} \wedge \neg c^{!\bullet} \Rightarrow (a^{!\bullet} \wedge b^{!\bullet})) \wedge (\neg a^{!\bullet} \wedge b^{!\bullet} \wedge (a^{!\circ} \vee b^{!\circ}) \Rightarrow c^{!\bullet})$
	$\psi_{Replicator}(a,b,c) : \tilde{a} \Leftrightarrow (\tilde{b} \wedge \tilde{c}) \wedge \neg \tilde{a} \Rightarrow ((\neg a^{\triangleright} \wedge b^{\triangleright}) \vee (\neg b^{\triangleright} \wedge c^{\triangleright}) \vee (\neg c^{\triangleright} \wedge b^{\triangleright} \wedge a^{\triangleright})) \wedge c^{\triangleright} \wedge a^{\triangleright})) \wedge (a^{!\circ} \wedge \neg a^{!\bullet} \Rightarrow (b^{!\bullet} \wedge c^{!\bullet})) \wedge (\neg b^{!\bullet} \wedge c^{!\bullet} \wedge (b^{!\circ} \vee c^{!\circ}) \Rightarrow a^{!\bullet})$
	$\psi_{Router}(a,b,c) : \tilde{a} \Leftrightarrow (\tilde{b} \vee \tilde{c}) \wedge \neg(\tilde{b} \wedge \tilde{c}) \wedge \tilde{a} \Leftrightarrow (\neg a^{\triangleright} \vee \neg(b^{\triangleright} \vee c^{\triangleright})) \wedge (a^{!\circ} \wedge \neg a^{!\bullet} \Rightarrow (b^{!\bullet} \wedge c^{!\bullet})) \wedge (\neg b^{!\bullet} \wedge c^{!\bullet} \wedge (b^{!\circ} \vee c^{!\circ}) \Rightarrow a^{!\bullet})$

In Chapter 6, we presented the constraints that a primitive imposes on a network as a CSP. Here 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:

$$\neg(a^{!\circ} \vee a^{!\bullet} \vee b^{!\circ} \vee b^{!\bullet}) \vee (a^{!\circ} \wedge \neg a^{!\bullet} \wedge b^{!\bullet}) \vee (a^{!\bullet} \wedge b^{!\circ} \wedge \neg b^{!\bullet}).$$

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

Table 7.2.1 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 7.2.4 (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:*

$$\forall b \in B : b^{!\circ} \Rightarrow b^{!\bullet}.$$

Definition 7.2.3 (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 7.2.4 (Composition $\boxdot$) *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:*

$$L_1 \boxdot L_2 = (\mathcal{N}_1 \cup \mathcal{N}_2,\ \mathcal{M}_1 \cup \mathcal{M}_2,\ \rightarrow,\ q_{0_1} \times q_{0_2})$$

where $\rightarrow$ is defined as:

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

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

and its symmetric rule.

We define few operations on a solution s for $\Psi = \langle \mathcal{N}_\Psi,\ \mathcal{M}_\Psi,\ M_{\Psi 0},\ \ \mathcal{V}_\Psi,\ C_\Psi \rangle$:

- source(s)=$\langle \{m | m^\circ \in \mathcal{M}_\Psi : s(m^\circ) = \top\} \rangle$,

- target(s)=$\langle \{m | m'^\circ \in \mathcal{M}_\Psi : s(m'^\circ) = \top\} \rangle$,

- flow(s)=$\{n | n \in \mathcal{N}_\Psi:\ s(\tilde{n}) = \top\}$,

- reason-giving(s)=$\{n | n \in \mathcal{N}_\Psi : s(n^\triangleright) = \top\}$,

- priority(s)=$\{n | n \in \mathcal{N}_\Psi : (s(n^{!^\circ}) \vee s(n^{!^\bullet})) = \top\}$.

We say $s \backsim q \xrightarrow{N,R,I} p$, where

- q = source(s),

- N = flow(s),

- R=reason-giving(s),

- I = priority(s),

- p = target(s).

Definition 7.2.5 (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*

- $\mathcal{M} = \{m | s(m^\circ) = \top \vee s(m'^\circ) = \top, s \in \mathfrak{S}(\Psi)\}$,

- $Q = \bigcup_{s\in\mathfrak{S}(\Psi)}\{source(s), target(s)\}$,

- $\rightarrow = \{(source(s), flow(s), \textit{reason-giving}(s), \textit{priority}(s), target(s)) \mid s \in \mathfrak{S}(\Psi)\}$,

- $q_0 = source(s_0)$.

Theorem 7.2.1 *Let* Ψ_1 *and* Ψ_2 *be two RCSPs, we show that* $\gamma(\Psi_1 \odot \Psi_2) = \gamma(\Psi_1) \boxdot \gamma(\Psi_2)$.

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 $\gamma(\Psi_1 \odot \Psi_2) = (\mathfrak{N}, Q, \rightarrow, q_0)$.

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 $\exists s \in \mathfrak{S}(\Psi_1 \odot \Psi_2)$, $s_1, \in \mathfrak{S}_1$, $s_2 \in \mathfrak{S}_2$, $t_1 : q_1 \xrightarrow{N_1,R_1,I_1}{}_1 p_1$, $t_2 : q_2 \xrightarrow{N_2,R_2,I_2}{}_2 p_2$ s.t. $s_1 \backsim t_1$ and $s_2 \backsim t_2$, but $\nexists\, t : q \xrightarrow{N,R,I} p \in \rightarrow$ s.t. $s \backsim t$.

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. $s \backsim t$.

RLTS is comparable with *Reo automata* [BCS12], a context-dependent formal semantics 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.

7.3 Numeric priority

Here, we extend our approach to support numeric priorities. This enables us to deal with more than one level of priorities such as in a process where the normal flow may be interrupted by both *exception* and *error*.

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.

$\langle\delta\rangle \models x \geq P$ 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 7.2.3.

Definition 7.3.1 (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 7.3.2 (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*

- $\mathcal{N}_L = \{n | s(\tilde{n}) = \top,\ s \in \mathfrak{S}(\Psi)\}$,
- $\mathcal{M}_L = \{m | s(m^\circ) = \top\ \vee\ s(m'^\circ) = \top,\ s \in \mathfrak{S}(\Psi)\}$,

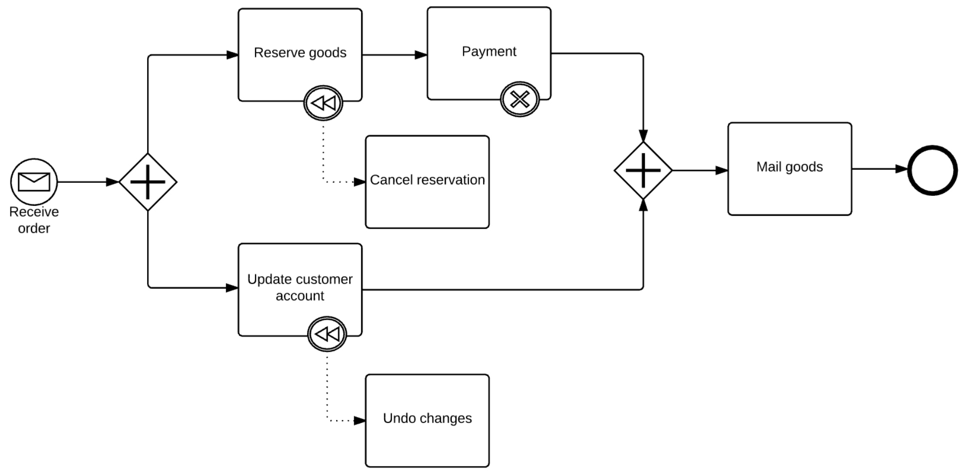


Figure 7.4.1: An example of a sales process modeled in BPMN

- $Q = \bigcup_{s \in \mathfrak{S}(\Psi)} \{source(s),\ target(s)\}$,
- $\rightarrow\ = \{(source(s),\ flow(s),\ reason\text{-}giving(s),\ priority(s),\ target(s)) \mid s \in \mathfrak{S}(\Psi)\}$, $q_0 = source(s_0)$.

7.4 Case study

In this section, we present the applications of our approach on a priority-aware model. Figure 7.4.1 depicts a sales process, which starts by receiving an order from a customer. It proceeds by reserving the ordered items for the customer. Then, the customer's credit gets charged and the customer's account is updated, meanwhile if the payment encounters a problem, a *cancellation* event is triggered, which causes compensation for any of the performed actions. Finally, if no problem occurs, the ordered items are shipped and the process ends.

Figure 7.4.2 shows a Reo network that simulates this process. Here, we use alphabet characters to refer to nodes (e.g. B, C) and channels (e.g. BC, BD). To address a node end or a channel end, we append a number to the name of an end, unless it is the only end (e.g. it is a boundary end). For instance, the end BC_2, which is the source end of the channel BC connects to the end B_2 on the node B. In [CKA10], the authors defined a procedure to map BPMN models to Reo networks.

The process starts by reading a token from the *writer* W_2, which resembles

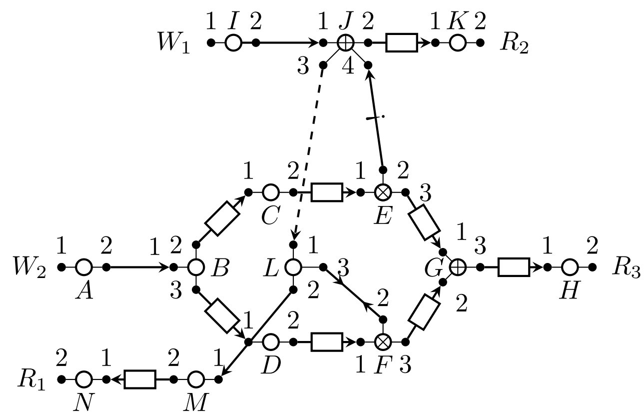


Figure 7.4.2: The process of a sample on-line shop modeled in Reo

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 duplicates it into the BC and BD $FIFO_1$ channels. The token continues to the CE $FIFO_1$ channel. If the payment succeeds, the token enters the EG $FIFO_1$ channel waiting for a token from the other input of the *merge* node G to enter the GH $FIFO_1$ channel and finally to be consumed by the *reader* R_3.

If the payment fails, performed actions need to be compensated. A token from the writer W_1 indicates a payment failure, so the process needs to be canceled. So, the token leaving the CE $FIFO_1$ channel goes through the EJ *prioritySync* channel. The *replicate* node J duplicates the token to the JK $FIFO_1$ and the JL *lossySync* channel. The *reader* R_2 consumes the token from the JK $FIFO_1$ channel, while the token from the JL *lossySync* channel moves forward to the MN $FIFO_1$ channel.

The token from the BD $FIFO_1$ channel goes through the DF $FIFO_1$ channel for a possible compensation. The token from the DF $FIFO_1$ channel may either go to the *join* node G to join the flow of a successful payment, or to be consumed by the LF *syncDrain*. In the latter case, it goes to the MN $FIFO_1$ channel. Then, the process ends by a read action of the *reader* R_1.

We compute the behavior of the given Reo network using our constraint-based framework. The steps for obtaining the RLTS are 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.

To show how priority can affect the behavior of 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* nodes E chooses one of its outgoing flows in a non-deterministic fashion.

We would like to check if for all transitions t, which belong to the RLTS of the network, the following holds: $\{CE,\ DF\} \subseteq source(t) \wedge E_1 \in flow(t) \wedge W_1 \notin reason-giving(t) \Rightarrow W_1 \in target(t)$. To violate this property, it is enough to find a transition from a state wherein both CE and DF $FIFO_1$ channels are full, there is flow on end E_1, W_1 is ready to communicate, but W_1 does not have flow.

Abstraction: For checking this assertion, we abstract from the ends without flow on transitions with the same source (q), target (p), ends with flow (N_1), but different ends without flow (N_2) by replacing them with $q \xrightarrow{N_1, N'_2} p$, where $N'_2 = \{W_1\}$ if $W_1 \in N_2$, otherwise $N'_2 = \{\}$. This abstraction reduces the number of transitions in the RLTS without affecting the result of the verification for the given assertion. We can take this one step further and remove the information about ends without flow from all the states except the state wherein CE and DF $FIFO_1$ channels are full.

Figure 7.4.3 shows the abstract (with respect to the given assertion) RLTS of the network of Figure 7.4.2 in absence of priority, where the transition t_4 violates the assertion. Here, we use short labels (e.g. t_4) on transitions and states. The original labels are represented in Table 7.4.1. In addition, the ends with a similar name are grouped e.g. $B_{1,2,3}$ (referring to ends B_1, B_2, and $B3$). This is only a presentation modification to save space. We show that the transition t_4 can not exist when the priority is considered in the model.

$$0 : \mathring{CE} \wedge \mathring{DF} \wedge \tilde{E}_1 \wedge \neg {W_1}^{\triangleright} \wedge \neg \tilde{W}_1 \ (the\ assertion)$$

$$1 : \frac{\Psi_{PrioritySync}(EJ_{2,4})}{EF_2^{!\bullet}}$$

$$2 : \frac{1 \text{ \&join of } EJ_2 \& E_2}{E_2^{!\bullet}}$$

$$3 : \frac{2; \Psi_{router}(E_{1,2,3})}{\tilde{E}_1 \wedge \neg \tilde{E}_2 \Rightarrow {E_2}^{\triangleright}}$$

$$4 : \frac{3 \text{ \& coloring \& join}}{{E_2}^{\triangleright} \Rightarrow {W_1}^{\triangleright}}$$

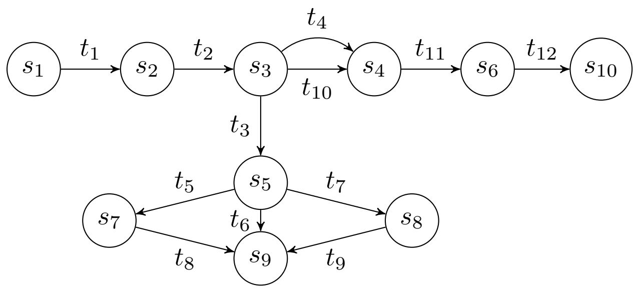


Figure 7.4.3: The RLTS corresponding to Reo network of Figure 7.4.2 with no priority channel

$$5 : \frac{2 \text{ \& } 4 \text{ coloring \& join}}{\neg\tilde{W}_1 \Rightarrow {W_1}^{\triangleright}}$$

$$6 : \frac{0\&5}{\perp}$$

7.5 Related work

Several works, e.g., [FPHA02, BK92, Bau97] use priorities to model scheduling policies. Many workflow languages rely on Petri nets [vdAtH02, YSSW08]. Priority flow in Petri net-based process models is managed with the help of inhibitor arcs and transition priorities [Pad15]. Inhibitor arcs allow a transition to fire only if the adjacent place is empty. *Prioritized Petri nets* [Bal01] 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., [LP16, RMP$^+$12] 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 [CPLA10]. It demonstrates the efficiency of the constraint-based approach. It models synchronization and data flow constraints, but no priority flow was considered. In [CKA12],

Table 7.4.1: The transition labels and prioritized ends (P) of the RLTS of Figure 7.4.3

s_1	$\langle\rangle$
s_2	$\langle BC, BD\rangle$
s_3	$\langle CE, DF\rangle$
s_4	$\langle EG, FG\rangle$
s_5	$\langle MN, JK\rangle$
s_6	$\langle GH\rangle$
s_7	$\langle JK\rangle$
s_8	$\langle MN\rangle$
s_9	$\langle\rangle$
s_{10}	$\langle\rangle$
t_1	$N_1 : \{W_2, A_{1,2}, B_{1,2,3}, AB_{1,2}, BC_2, BD_3\}, N_2 : \{\}$
t_2	$N_1 : \{BC_1, BD_1, C_{1,2}, D_{1,2}, CE_2, DF_2\}, N_2 : \{\}$
t_3	$N_1 : \{W_1, CE_2, DF_3, IJ_{2,3}, I_{1,2}, J_{1,2,3,4}, JK_2, JL_{1,3},$ $L_{1,2,3}, LF_{2,3}, LM_{1,2}, F_{1,2}, M_{1,2}, MN_2\}, N_2 : \{\}$
t_4	$N_1 : \{EG_3, FG_3, E_{1,3}, F_{1,3}, CE_1, DF_1\},$ $N_2 : \{W_1\}$
t_5	$N_1 : \{R_1, N_{1,2}, MN_1\}, N_2 : \{\}$
t_6	$N_1 : \{R_{1,2}, N_{1,2}, MN_1, K_{1,2}, JK_1\}, N_2 : \{\}$
t_7	$N_1 : \{R_2, K_{1,2}, JK_1\}, N_2 : \{\}$
t_8	$N_1 : \{R_2, K_{1,2}, JK_1\}, N_2 : \{\}$
t_9	$N_1 : \{R_1, N_{1,2}, MN_1\}, N_2 : \{\}$
t_{10}	$N_1 : \{EG_3, FG_3, E_{1,3}, F_{1,3}, CE_1, DF_1\},$ $N_2 : \{W_1\}$
t_{11}	$N_1 : \{EG_1, FG_2, G_{1,2,3}, GH_3\}, N_2 : \{\}$
t_{12}	$N_1 : \{R_3, H_{1,2}, GH_1\}, N_2 : \{\}$
P	$\{W_1, I_{1,2}, J_{1,2,3,4}, JK_2, EJ_{2,4}, E_{1,2}, JL_{1,3}, L_{1,2,3},$ $LF_{2,3}, F_{2,3}, LM_{1,2}, M_{1,2}, MN_2\}$

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 tool-set 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 [KAT16], 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.

7.6 Conclusions and future work

In this chapter, 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. Furthermore, we showed correctness of our approach for the binary case. 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.

8
Conclusion

Despite long-term efforts, analyzing business processes is still a challenge. Creating tools for analyzing business processes requires expressing the behavior of the processes in an accurate way. Most of the business process management notations, particularly Business Process Model and Notation (BPMN), are based on Petri nets.

While Petri nets can be used to automate process analysis, they are not compositional. This makes analyzing the behavior of large and complex models based on Petri nets challenging.

The Reo coordination language is an alternative theory to Petri nets that has been used to formalize semantics of BPMN. Reo has a compositional nature, which enables adding semantic models for individual components to the semantic models of existing processes.

In this dissertation, we used the Reo coordination language to capture the behavior of BPMN processes. We presented an automated mapping of business process models expressed in BPMN 2 to Reo networks in order to create the possibility of using various types of analysis on business process models. Our mapping takes data into account. Thus, it enables verification of data flow. We not only deal with basic BPMN 2 constructs, but also with compound elements such as transactions and exception handling. Formalizing the behavior of these elements requires modeling priority.

Reo is an extensible language that comes with various formal semantic models. This makes it possible to perform different kinds of analysis by focusing on specific behavioral aspects of a given network. However, there is a gap between the behavior that each of the semantics can express. This can introduce incompatibilities among these operational models. In addition, these formal semantics are computed using their own specialized algorithms, which are directly implemented.

Such algorithms are computationally expensive. As a result, the Reo models (and consequently business models) whose operational semantics can be efficiently calculated are limited to those of relatively small size.

Each of these formal semantics constrain the possible I/O operations through the nodes to those allowed by the semantics. Therefore, we convert the problem of finding behaviors accepted by a given semantic model into a constraint satisfaction problem for which many efficient supporting tools exist.

We developed a unified constraint-based framework to compute formal semantics of a Reo network given the semantics of its parts in a compositional fashion. Since we have included various existing formal semantics of Reo in our framework, behavior specifications that are considered invalid according to any of these formal semantics are ruled out. The tool we implemented to realize this framework relies on constraint solvers. Therefore, it benefits from the advances in the field of constraint solving.

Within this framework, the behavior of a Reo construct specified by a given semantics model is expressed in terms of constraints. In order to obtain the semantics of the whole Reo connector, the constraints of its constructs are concatenated. The framework replaces data constraints with new binary predicates that represent the logical value of the data constraint. The final constraint is then converted to the acceptable format for an off-the-shelf constraint solver.

After the constraint solver finds the solutions, the solutions are mapped back to the predicates. The data constraints and the value of their representative predicate are sent to a numeric constraint solver that treats the data symbolically. This way instead of obtaining distinct possible values for each variable denoting a data-item, we have a range of values, which is a more compact representation. We compared the performance of our approach to the existing ways of computing the formal semantics of Reo.

We presented a constraint-based approach for calculating priority-aware semantics of Reo models. This approach has been integrated into the mentioned constraint-based framework as the first tool support for priority in Reo. Similarly, this approach benefits from the shift of paradigm from custom direct implementation to using tools available in the well researched area of constraint solving. We not only provide a way to model the binary notion of priority in Reo, but also we deal with numeric priority. We demonstrated the application of our toolchain by analyzing a BPMN process that could not be analyzed previously.

A limitation of our implemented toolchain is that it relies on the external BPMN modeling tools to create the BPMN process to be analyzed. Since not all BPMN tools support export the BPMN models in our expected format, the choice of BPMN editor compatible with our tool set is limited.

As our future work, we plan to expand our constraint-based semantics framework to include other formal semantics of Reo, for instance, those that incorporate stochastic and quantitative aspects of the behavior of Reo circuits. In addition, we plan to extend our constraint-based framework to generate data to be used for simulation and testing purposes.

Bibliography

[AAA$^+$09] Bernhard Aichernig, Farhad Arbab, Lacramioara Astefanoaei, Frank S. de Boer, Sun Meng, and Jan J. M. M. Rutten. Fault-based Test Case Generation for Component Connectors. In *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*, pages 147–154. IEEE, july 2009.

[ABBR04] Farhad Arbab, Christel Baier, Frank De Boer, and Jan J. M. M. Rutten. Models and Temporal Logics for Timed Component Connectors. In *2nd International Conference on Software Engineering and Formal Methods*, pages 198–207. IEEE Computer Society, 2004.

[ABC$^+$09] Farhad Arbab, Roberto Bruni, Dave Clarke, Ivan Lanese, and Ugo Montanari. Tiles for Reo. In *Recent Trends in Algebraic Development Techniques*, volume 5486 of *Lecture Notes in Computer Science*, chapter 4, pages 37–55. Springer, 2009.

[Abd02] Abdelwaheb Ayari and David Basin. QUBOS: Deciding Quantified Boolean Logic using Propositional Satisfiability Solvers. In *Formal Methods in Computer-Aided Design*, pages 187–201. Springer, 2002.

[ABS15] Farhad Arbab, Christel Baier, and Marjan Sirjani. Priority in Reo and Constraint Automata. Technical report, Centrum voor Wiskunde en Informatica, 2015. In preparation.

[ACMM07] Farhad Arbab, Tom Chothia, Sun Meng, and Young-Joo Moon. Component Connectors with QoS Guarantees. In *Coordination Models and Languages, 9th International Conference*, pages 286–304, 2007.

[act] Activiti. `http://www.activiti.org`. Accessed: 2019-09-30.

[ACvdM$^+$09] Farhad Arbab, Tom Chothia, Rob van der Mei, Sun Meng, Youngjoo Moon, and Chretien Verhoef. From coordination to stochastic models of QoS. In *Coordination Models and Languages*, pages 268–287. Springer, 2009.

[AJ15] Farhad Arbab and Sung-Shik T. Q. Jongmans. Coordinating Multicore Computing. In *Advanced Lectures of the 15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Formal Methods for Multicore Programming*, volume 9104 of *Lecture Notes in Computer Science*, pages 57–96. Springer International Publishing, 2015.

[AKM$^+$08a] Farhad Arbab, Christian Koehler, Ziyan Maraikar, Young-Joo Moon, and José Proença. Modeling, Testing and Executing Reo Connectors with the Eclipse Coordination Tools. In *5th International Workshop on Formal Aspects of Component Software*, volume 8. ENTCS, 2008.

[AKM08b] Farhad Arbab, Natallia Kokash, and Sun Meng. Towards Using Reo for Compliance-Aware Business Process Modeling. In *ISoLA*, pages 108–123, 2008.

[AM08] Farhad Arbab and Sun Meng. Synthesis of Connectors from Scenario-based Interaction Specifications. In *Proceedings of the International Symposium on Component Based Software Engineering*, volume 5282 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2008.

[AP08] Ahmed Awad and Frank Puhlmann. Structural detection of deadlocks in business process models. In *Business Information Systems, 11th International Conference*, pages 239–250, 2008.

[AR02] Farhad Arbab and Jan J. M. M. Rutten. A coinductive calculus of component connectors. In *Recent Trends in Algebraic Development Techniques*, volume 2755 of *Lecture Notes in Computer Science*, pages 35–56. Springer-Verlag, 2002.

[Arb02] Farhad Arbab. Abstract Behavior Types: A Foundation Model for Components and Their Composition. In *Formal Methods for Components and Objects*, pages 33–70, 2002.

[Arb04] Farhad Arbab. Reo: a Channel-Based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14:329–366, 2004.

[Arb06] Farhad Arbab. Computing and interaction. In D. Goldin, S. Smolka, and P. Wegner, editors, *Composition of Interacting Computations*, pages 277–321. Springer-Verlag, 2006.

[ari] ARIS Express. `http://www.ariscommunity.com/aris-express`. Accessed: 2019-09-30.

[Bai05] Christel Baier. Probabilistic Models for Reo Connector Circuits. *Journal of Universal Computer Science*, 11(10):1718–1748, 2005.

[Bal01] Gianfranco Balbo. *Introduction to Stochastic Petri Nets*, pages 84–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[Bau97] Falko Bause. *Analysis of Petri nets with a dynamic priority method*, pages 215–234. Springer, 1997.

[BBK$^+$10] Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolfgang Leister. Design and verification of systems with exogenous coordination using vereofy. In *ISoLA (2)*, volume 6416 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2010.

[BCS12] Marcello Bonsangue, Dave Clarke, and Alexandra Silva. A Model of Context-dependent Component Connectors. *Science of Computer Programming*, 77(6):685 – 706, 2012.

[BFV11] Kelly Rosa Braghetto, Joao Eduardo Ferreira, and Jean-Marc Vincent. From Business Process Model and Notation to Stochastic Automata Network. Research report, Universidade Sao Paulo, 2011.

[BHF05] Michael Butler, Tony Hoare, and Carla Ferreira. A Trace Semantics for Long-running Transactions. In *Proceedings of the International Conference on Communicating Sequential Processes: The First 25 Years*, CSP'04, 2005.

[BJT05] Jean Bézivin, Frédéric Jouault, and David Touzet. An introduction to the atlas model management architecture. Technical Report 05-01, LINA, 2005.

[BK92] Eike Best and Maciej Koutny. Petri net Semantics of Priority Systems. *Theoretical Computer Science*, 96(1):175 – 215, 1992.

[BPM] BPMN 2.0 Modeler. `https://www.eclipse.org/proposals/soa.bpmn2-modeler/`. Accessed: 2019-09-30.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling Component Connectors in Reo by Constraint Automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. *Handbook of Satisfiability*, 4, 2009.

[BW06] Christel Baier and Verena Wolf. Stochastic Reasoning about Channel-based Component Connectors. In *Coordination Models and Languages*, volume 4038 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2006.

[BZ07] Lucas Bordeaux and Lintao Zhang. A Solver for Quantified Boolean and Linear Constraints. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 321–325. ACM, 2007.

[CCA07] Dave Clarke, David Costa, and Farhad Arbab. Connector Colouring I: Synchronisation and Context Dependency. *Science of Computer Programming*, 66(3):205–225, 2007.

[CCH11] David Raymond Christiansen, Marco Carbone, and Thomas Hildebrandt. Formal Semantics and Implementation of BPMN 2.0 Inclusive Gateways. In *Web Services and Formal Methods*, page 146–160. Springer, 2011.

[CKA10] Behnaz Changizi, Natallia Kokash, and Farhad Arbab. A Unified Toolset for Business Process Model Formalization. In *7th International Workshop on Formal Engineering approaches to Software Components and Architectures*, pages 147–156. ENTCS, 2010.

[CKA12] Behnaz Changizi, Natallia Kokash, and Farhad Arbab. A Constraint-based Method to Compute Semantics of Channel-based Coordination Models. In *International Conference on Software Engineering Advances*. IARA, 2012.

[CKA19] Behnaz Changizi, Natallia Kokash, and Farhad Arbab. Service Orchestration with Priority Constraints. In *International Conference on Software Engineering Advances*, Lecture Notes in Computer Science. Springer-Verlag, 2019.

[CM02] Manuel Clavel and José Meseguer. Reflection in Conditional Rewriting Logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

[Cos10] David Costa. *Formal Models for Context Dependent Connectors for Distributed Software Components and Services*. PhD thesis, Vrij Universiteit Amsterdam, 2010.

[CPLA10] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, In Press, Accepted Manuscript, 2010.

[Dav88] James H. Davenport. Computer Algebra Applied to Itself. *Journal of Symbolic Computation*, 6:127–132, 1988.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Information & Software Technology*, 50(12):1281–1294, 2008.

[DM03] Thomas Dufresne and James Marti. *Process Modeling for E-Business*. INFS 770 Methods for Information Systems Engineering: Knowledge Management and E-Business., 2 edition, 2003.

[DRMR13] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013.

[DW11] Gero Decker and Mathias Weske. Interaction-centric Modeling of Process Choreographies. *Information Systems*, 36(2):292–312, 2011.

[ESB14] Nissreen El-Saber and Artur Boronat. BPMN Formalization and Verification Using Maude. In *Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications*, BM-FA '14, pages 1:1–1:12. ACM, 2014.

[fla] Adobe Flash. `https://get.adobe.com/flashplayer/`. Accessed: 2019-09-30.

[FPHA02] Reinhard Füricht, Herbert Prähofer, Thomas Hofinger, and Josef Altmann. A Component-based Application Framework for Manufacturing Execution Systems in C# and .NET. pages 169–178, 2002.

[GLS17] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. From LOTOS to LNT. *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, page 3–26, 2017.

[GMR+06] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. The Formal Specification Language mCRL2. In *Methods for Modelling Software Systems*, volume 06351 of *Dagstuhl Seminar Proceedings*. IBFI, 2006.

[GPR+07] Jan F. Groote, Marija Petkovic, Ivo Raedts, Yaroslav S. Usenko, Lou J. Somers, and Jan Martijn E.M. van der Werf. Transformation of BPMN Models for Behaviour Analysis. In *Proceedings of the 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*, pages 126–137. INSTICC Press, 2007.

[Gro11] Object Management Group. Business Process Model and Notation (BPMN) Version 2.0. Technical report, Object Management Group, 2011.

[HHL+12] Monika Heiner, Mostafa Herajy, Fei Liu, Christian Rohr, and Martin Schwarick. Snoopy – A Unifying Petri Net Tool. In *Application and Theory of Petri Nets*, page 398–407. Springer, 2012.

[Hoa85] Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[Hoa13] Tony Hoare. Unifying Semantics for Concurrent Programming. In *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky - Essays Dedicated to Samson Abramsky on the Occasion of His 60th Birthday*, volume 7860 of *Lecture Notes in Computer Science*, pages 139–149. Springer, 2013.

[Hoo11] Geoffrey Hook. Business process modeling and simulation. In *Winter Simulation Conference*, pages 773–778, 2011.

[IB08] Mohammad Izadi and Marcello M. Bonsangue. Recasting Constraint Automata into Büchi Automata. In *International Colloquium on Theoretical Aspects of Computing*, pages 156–170, 2008.

[IBC08] Mohammad Izadi, Marcello Bonsangue, and Dave Clarke. Modeling Component Connectors: Synchronisation and Context-dependency. In *Software Engineering and Formal Methods*, pages 303–312. IEEE Press, 2008.

[IBC11] Mohammad Izadi, Marcello Bonsangue, and Dave Clarke. Büchi Automata for Modeling Component Connectors. *Software and System Modeling*, 10(2):183–200, 2011.

[JA12] Sung-Shik T.Q. Jongmans and Farhad Arbab. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science*, 22:201–251, 2012.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*, pages 128–138, 2005.

[JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.

[JSS+12] Sung-Shik T. Q. Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. *Proceedings of the International Conference on Service-Oriented and Cloud Computing*, pages 1–16. Springer, 2012.

[KA13] Natallia Kokash and Farhad Arbab. Formal Design and Verification of Long-Running Transactions with Extensible Coordination Tools. *IEEE Trans. Serv. Comput.*, 6(2):186–200, 2013.

[Kan10] Oscar Kanters. QoS Analysis by Simulation in Reo. diploma thesis, CWI Amsterdam and Vrije Universiteit Amsterdam, 2010, 2010.

[KAT16] Tobias Kappé, Farhad Arbab, and Carolyn L. Talcott. A Compositional Framework for Preference-Aware Agents. In *Proceedings of The First Workshop on Verification and Validation of Cyber-Physical Systems*, pages 21–35, 2016.

[KB09] Sascha Klüppelholz and Christel Baier. Symbolic Model Checking for Channel-based Component Connectors. *Science of Computer Programming*, 74(9):688 – 701, 2009.

[KC09] Christian Koehler and Dave Clarke. Decomposing Port Automata . In *SAC'09: Proceedings of 2009 ACM Symposium on Applied Computing*, pages 1369–1373. ACM, 0 2009.

[KCA10] Natallia Kokash, Behnaz Changizi, and Farhad Arbab. A Semantic Model for Service Composition with Coordination Time Delays. In *ICFEM*, pages 106–121, 2010.

[KKdV10] Natallia Kokash, Christian Krause, and Erik de Vink. Data-aware Design and Verification of Service Compositions with Reo and mCRL2. In *SAC'10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2406–2413. ACM, 0 2010.

[KMLA11] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling Dynamic Reconfigurations in Reo Using High-level Replacement Systems. *Science of Computer Programming*, 76(1):23 – 36, 2011.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic Symbolic Model Checker. In *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference*, pages 200–204, 2002.

[Kra11] Christian Krause. *Reconfigurable Component Connectors*. PhD thesis, Leiden University, The Netherlands, 0 2011.

[LP16] Irina. A. Lomazova and Louchka Popova-Zeugmann. Controlling Petri Net Behavior using Priorities for Transitions. *Fundamenta Informaticae*, 143(1-2):101–112, 2016.

[LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A Calculus for Orchestration of Web Services. In *Programming Languages and Systems*, page 33–47. Springer Berlin Heidelberg, 2007.

[MA07a] Sun Meng and Farhad Arbab. On Resource-Sensitive Timed Component Connectors. In *Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *LNCS*, pages 301–316. Springer, 2007.

[MA07b] Sun Meng and Farhad Arbab. Web Services Choreography and Orchestration in Reo and Constraint Automata. In *Proceedings of the ACM Symposium on Applied Computing*, pages 346–353. ACM Press, 2007.

[MA09] Sun Meng and Farhad Arbab. QoS-Driven Service Selection and Composition Using Quantitative Constraint Automata. *Fundamenta Informaticae*, 95(1):103–128, 2009.

[MA10] Sun Meng and Farhad Arbab. A Model for Web Service Coordination in Long-Running Transactions. In *The Fifth IEEE International Symposium on Service-Oriented System Engineering*, pages 121–128, 2010.

[MAA+12] Sun Meng, Farhad Arbab, Bernhard K. Aichernig, Lacramioara Astefanoaei, Frank S. de Boer, and Jan J. M. M. Rutten. Connectors as Designs: Modeling, Refinement and Test Case Generation. *Science of Computer Programming*, 77(7-8):799–822, 2012.

[Mar09] Marcello M. Bonsangue and Dave Clarke and Alexandra Silva. Automata for Context-Dependent Connectors. In *COORDINATION*, pages 184–203, 2009.

[MBL+18] Umair Mutarraf, Kamel Barkaoui, Zhiwu Li, Naiqi Wu, and Ting Qu. Transformation of Business Process Model and Notation Models onto Petri nets and Their Analysis. *Advances in Mechanical Engineering*, 10(12):1687814018808170, 2018.

[MSA04] Mohammad Reza Mousavi, Marjan Sirjani, and Farhad Arbab. Specification and Verification of Component Connectors. Technical Report CSR-04-15, Department of Computer Science, Eindhoven University of Technology, 2004.

[MSA06] Mohammad Reza Mousavi, Marjan Sirjani, and Farhad Arbab. Formal Semantics and Analysis of Component Connectors in Reo. *Electronic Notes in Theoretical Computer Science*, 154(1):83 – 99, 2006.

[MSKA10] Young-Joo Moon, Alexandra Silva, Christian Krause, and Farhad Arbab. A Compositional Semantics for Stochastic Reo Connectors. In *Proceedings Ninth International Workshop on the Foundations of Coordination Languages and Software Architectures*, pages 93–107, 2010.

[MSKA14] Young-Joo Moon, Alexandra Silva, Christian Krause, and Farhad Arbab. A compositional model to reason about end-to-end QoS in Stochastic Reo connectors. *Science of Computer Programming*, 80:3–24, 2014.

[MSTV07] Roshanak Zilouchian Moghaddam, Marjan Sirjani, Samira Tasharofi, and Mohsen Vakilian. Modeling Web Service Interactions using the Coordination Language Reo. In *Proceedings of the 4th International Workshop on Web Services and Formal Methods*, volume 4937 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2007.

[MSY14] Radu Mateescu, Gwen Salaün, and Lina Ye. Quantifying the parallelism in bpmn processes using model checking. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 159–168. ACM, 2014.

[PA91] Brigitte Plateau and Karim Atif. Stochastic Automata Network For Modeling Parallel Systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

[Pad15] Julia Padberg. Reconfigurable Petri Nets with Transition Priorities and Inhibitor Arcs. *The Proceedings of the 8th International Conference on Graph Transformation*, pages 104–120, 2015.

[Plo04] Gordon Plotkin. A Structural Approach to Operational Semantics. *The Journal of Logic and Algebraic Programming*, 60–61(0):17 – 139, 2004.

[PQZ08] Davide Prandi, Paola Quaglia, and Nicola Zannone. Formal Analysis of BPMN Via a Translation into COWS. In *Coordination Models and*

Languages, 10th International Conference, COORDINATION, pages 249–263, 2008.

[Pro11] José Proença. *Synchronous Coordination of Distributed Components.* PhD thesis, Institue for Prgramming research and Algorithms, 2011.

[PS12] Pascal Poizat and Gwen Salaün. Checking the Realizability of BPMN 2.0 Choreographies. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1927–1934. ACM, 2012.

[PSHA12] Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat, and Farhad Arbab. Symbolic Execution of Reo Circuits using Constraint Automata. *Science of Computer Programming*, 77(7-8):848–869, 2012.

[Ray87] Gerhard Rayna. *REDUCE: Software for Algebraic Computation.* Springer-Verlag New York, Inc., 1987.

[RBM05] Ugo Roberto Bruni, Hernán C. Melgratti and Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 2005.

[RKNP04] Jan J. M. M. Rutten, Marta Kwiatkowska, Gethin Norman, and David Parker. *Component Connectors*, volume 23 of *CRM Monograph Series.* American Mathematical Society, 2004.

[RMP+12] Valentín Valero Ruiz, Hermenegilda Macià, Juan José Pardo, María-Emilia Cambronero, and Gregorio Díaz. Transforming Web Services Choreographies with priorities and time constraints into prioritized-time colored Petri nets. *Science of Computer Programming*, 77(3):290 – 313, 2012.

[Rol95] Asbjorn Rolstadas. Business Process Modeling and Reengineering. In *Performance Management: A Business Process Benchmarking Approach*, pages 148–150, 1995.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

[SOP+06] Lou J. Somers, Olivia Oanea, Reinier Post, Kees M. van Hee, and Jan Martijn E. M. van der Werf. Yasper: A Tool for Workflow Modeling and Analysis. In *Sixth International Conference on Application of Concurrency to System Design*, volume 00, pages 279–282, 2006.

[STK+10] David Schumm, Oktay Turetken, Natallia Kokash, Amal Elgammal, Frank Leymann, and Willem-Jan van den Heuvel. Business Process Compliance through Reusable Units of Compliant Processes. In *Current Trends in Web Engineering: 10th Int. Conf. on Web Engineering*, pages 325–337. Springer, 2010.

[TJ10] Nasi Tantitharanukul and Watcharee Jumpamule. Detection of Live-Lock in BPMN Using Process Expression. volume 114 of *Communications in Computer and Information Science*, pages 164–174. Springer, 2010.

[TSJ10] Nasi Tantitharanukul, Prompong Sugunnasil, and Watcharee Jumpamule. Detecting Deadlock and Multiple Termination in BPMN Model using Process Automata. *ECTI-CON2010: The 2010 ECTI International Confernce on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pages 478–482, 2010.

[vdA98] Wil M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[vdA04] Wil M. P. van der Aalst. *Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management*, pages 1–65. Lecture Notes in Computer Science. Springer, 2004.

[vdADK02] Wil M. P. van der Aalst, Jörg Desel, and Ekkart Kindler. On the Semantics of EPCs: A Vicious Circle. In *EPK 2002 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings des GI-Workshops und Arbeitskreistreffens*, pages 71–79, 2002.

[vdAtH02] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. Technical Report DAIMI PB-560, 2002.

[vDdMV+05] Boudewijn F. van Dongen, Ana Karla A. de Medeiros, Henricus M. W. Verbeek, Ajmm Ton Weijters, and Wil M. P. van der Aalst. The ProM Framework: A New Era in Process Mining Tool Support. In *Applications and Theory of Petri Nets*, page 444–454. Springer, 2005.

[VvdAK04] Henricus M. W. Verbeek, Wil M. P. van der Aalst, and Akhil Kumar. XRL/Woflan: Verification and Extensibility of an XML/Petri-Net-Based Language for Inter-Organizational Workflows. *Information Technology and Management*, 5(1-2):65–110, 2004.

[WG08] Peter Y.H. Wong and Jeremy Gibbons. A Process Semantics for BPMN. In *Proceedings of 10th International Conference on Formal Engineering Methods.*, volume 5256 of *Lecture Notes in Computer Science*, page 355–374, 2008.

[Wil67] Stanley Williams. Business Process Modeling Improves Administrative Control. *Automation*, pages 44–50, 1967.

[Yao] Yaoqiang BPMN Editor. `http://sourceforge.net/projects/bpmn/`.

[YSSW08] JianHong Ye, ShiXin Sun, Wen Song, and LiJie Wen. Formal Semantics of BPMN Process Models Using YAWL. *Intelligent Information Technology Applications*, 2:70–74, 2008.

Summary

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.

Business Process Model and Notation (BPMN) has become the de-facto standard for business processes diagrams. In order to provide tools support to analyze the behavior of a BPMN model, in this dissertation, we present a mapping of BPMN models to Reo networks. The Reo coordination language is an exogenous coordination language that realizes the coordination patterns in terms of its complex networks, that are built out of simple primitives called channels. The mapping of BPMN to Reo is implemented as a plugin in the Reo analysis tool-set in a model-driven paradigm. Our mapping covers not only basic BPMN constructs but also advanced structures such as BPMN transactions.

Reo is easily extensible to support more advanced process models by defining new channels. However, the open-ended nature of Reo channels makes 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 that vary from rather simple that cover basic Reo behavior to more complex models that capture specific behavioral aspects, e.g., context-sensitivity. In some of these semantic models, computing the overall semantics of a system given semantics for its parts is computationally expensive. This hampers using the language for analyzing large real-world business processes.

In this dissertation, we present a constraint-based framework, which unifies various formal semantics of Reo. In this framework, the behavior of a Reo network is described using constraints. The constraint-based nature of our approach allows the simultaneous coexistence of several semantics in a simple fashion. The behavior of a Reo network is determined by the solutions to these constraints. Since any solution should satisfy all the encoded formal semantics, the framework eliminated any inconsistent behavior between the Reo formal semantics.

Another advantage of our proposed constraint-based approach compared to the existing approaches of calculating formal semantics of Reo is its efficiency due to efficient constraint solving methods and optimization techniques that are used in the off-the-shelf constraint solvers. We support this claim with a case study.

Among the behavioral aspects required to model a business process is priority. The notion of priority is necessary for modeling behaviors such as transaction and exception handling, where the data flow representing the error or exception should interrupt the normal flow.

In this dissertation, we present an alternative approach to model priority in Reo by extending our constraint-based framework with priority-aware premises. Further, we extend our priority-aware formal model to support not only a binary notion of priority, but also numeric priorities.

Samenvatting (Dutch Summary)

Bedrijfsprocesbeheer is een operationele managementaanpak die zich richt op het verbeteren van bedrijfsprocessen. Bedrijfsprocessen, d.w.z. verzamelingen van belangrijke activiteiten in een organisatie, worden weergegeven in de vorm van een workflow, een georkestreerd en herhalend patroon van activiteiten die geschikt zijn voor geautomatiseerde analyse en controle.

Business Process Model and Notation (BPMN) is de algemene standaard geworden voor bedrijfsprocesdiagrammen. Om ondersteunening in het analyseren van het gedrag van een BPMN-model, presenteren we in dit proefschrift een vertaling van BPMN-modellen naar Reo-netwerken. De Reo-coördinatietaal is een exogene coördinatietaal die de coördinatiepatronen opnieuw benoemt in termen van complexe netwerken, die zijn opgebouwd uit simpele primitieven genaamd channels. De vertaling van BPMN naar Reo is geïmplementeerd als een Reo-analysetool in een modelgedreven paradigma. Onze vertaling omvat niet alleen standaard BPMN-constructies, maar ook geavanceerde structuren zoals BPMN-transacties.

Reo is eenvoudig uitbreidbaar om meer geavanceerde procesmodellen te ondersteunen door het definiëren van nieuwe channels. Het flexibele karakter van Reo-kanalen maakt het echter noodzakelijk om de formele semantiek van Reo uit te breiden met een aantal nieuwe concepten. Verschillende variaties van semantische modellen voor Reo voorgesteld, varierend van vrij eenvoudig en die betrekking hebben op het basisgedrag van Reo, tot meer complexe modellen die specifieke gedragsaspecten vast leggen, bijvoorbeeld contextgevoeligheid.

In vele van deze semantische modellen is de berekening van de algehele semantiek van het systeem gegeven de semantiek van zijn onderdelen is rekenkundig duur. Dit bemoeilijkt het gebruik van de taal voor het analyseren van grote bedrijfsprocessen. In dit proefschrift presenteren we een op constraint-based framework dat verschillende formele semantiek van Reo.

In dit framework wordt het gedrag van een Reonetwork beschreven met het behulp van constraints. Onze constraint-based framework aanpak maakt gelijktijdige bestaan van verschillende semantiek op een eenvoudige manier mogelijk. Het gedrag van een Reo-netwerk wordt bepaald door de oplossingen voor deze constraints. Aangezien elke oplossing zou moeten voldoen aan alle gecodeerde formele semantiek, elimineerde het elk inconsistent gedrag tussen de Reo-formele semantieken.

Een ander voordeel van onze voorgestelde constraint-based benadering vergeleken met de bestaande benaderingen van het berekenen van de formele semantiek van Reo is de efficiëntie door efficiënte constraint-solving methoden en optimalisatietechnieken die worden gebruikt in de off-the-shelf constraint-solvers. We ondersteunen deze bewering met een casestudy.

Onder de gedragsaspecten die vereist zijn om een bedrijfsproces te modelleren, is prioriteit. Prioriteit is nodig voor het modelleren van gedrag zoals transactions en exception handling, waarbij de dataflow die de error of exception representeert de normale flow moet onderbreken.

In dit proefschrift presenteren we een alternatieve benadering om prioriteit te modelen in Reo door onze framework uit te breiden met prioriteit. Bovendien breiden we ons prioriteitsbewust formele model uit om niet alleen binaire prioriteit, maar ook numerieke prioriteiten te ondersteunen.

About the author

Beehnaz Changizi was born on March 21st, 1979 in Hamedan, Iran. She completed her bachelor studies in Computer Engineering at the Faculty of Computer Engineering Amirkabir University of Technology - Tehran Polytechnic Tehran, Iran, in 2003. She has worked for several years as a software developer before starting a master's degree. She obtained her master of science in Software Engineering from Sharif University of Technology in 2007. In 2008, Behnaz moved to Amsterdam to become a Ph.D. student at the Leiden University as part of the COMPAS project, under the supervision of Prof. Dr. Farhard Arbab. After four years of being a full-time Ph.D. student, Behnaz returned to industry to follow her passion for creating software, while she continued working on her thesis.

Acknowledgement

I would like to express my gratitude to all the people who helped me in any form during the period of my Ph.D. It is impossible to name everyone here, but I would like to mention some of them.

I would like to thank my colleagues in CWI Jose Proenca, Michiel Helvensteijn, Joost Winter, Ziyan Maraikar, Young-Joo Moon, Lacramioara Astefanoaei, Helle Hansen, Christian Krause, Mahdi Jaghoori, Sun Meng, Stijn de Gouw, and Sung-Shik Jongmans, Alexandra Silva, Stephanie Kemper, Yunes Hassen, and Ivan Zapreev.

I would like to especially thank Bahareh Changizi for drawing the bird on the thesis cover, Iona Michaelis for editing the last chapter of my thesis, Davey Bruns for editing the Dutch version of my thesis's summary, and Michiel Helvensteijn for proofreading my thesis.

I give special thanks to my family and friends for lifting me: Bahareh Changizi, Behnam Changizi, Arash Malayeri, Narges Javaheri, Aylar Soltani, Amaneh Mahboubi, Somayeh Bakhtiari, Martina Chirilus-Bruckner, Naser Ayat, Anna Kovacs, Zaklina Stevic, Pedram Malayeri, and Reinahneh Zolfaghari.

This doctoral dissertation is dedicated to my parents to whom I am forever grateful for their love, support, and encouragement and to my husband Rory Michaelis for his love, support, and patience. He deserves the final acknowledgment as he suffered most from its completion.

Titles in the IPA Dissertation Series since 2017

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis* *of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems*. Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations*. Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries*. Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

A. Serrano Mena. *Type Error Customization for Embedded Domain-Specific Languages*. Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers*. Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems*. Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities*. Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis*. Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing*. Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Cosimulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04