Constraint-based analysis of business process models

Changizi, B.

Citation

Changizi, B. (2020, February 21). *Constraint-based analysis of business process models*. *IPA Dissertation Series*. Retrieved from https://hdl.handle.net/1887/85677

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/85677

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/85677 holds various files of this Leiden University dissertation.

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

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.

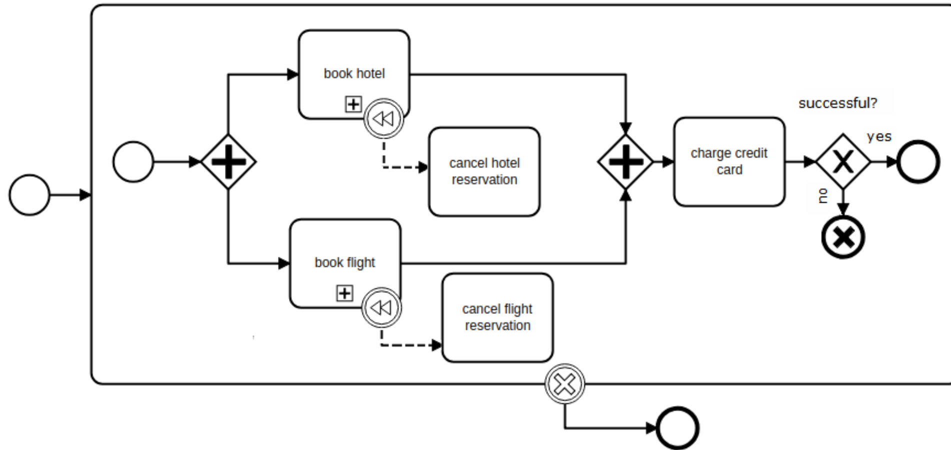
The handleTaskCompletion method creates a send signal event and places it after the given compensatable task (using a newly created gateway to connect it to the other elements if it is needed). Additionally, it creates a receive signal event to catch the generated signal event and adds it to the receivers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

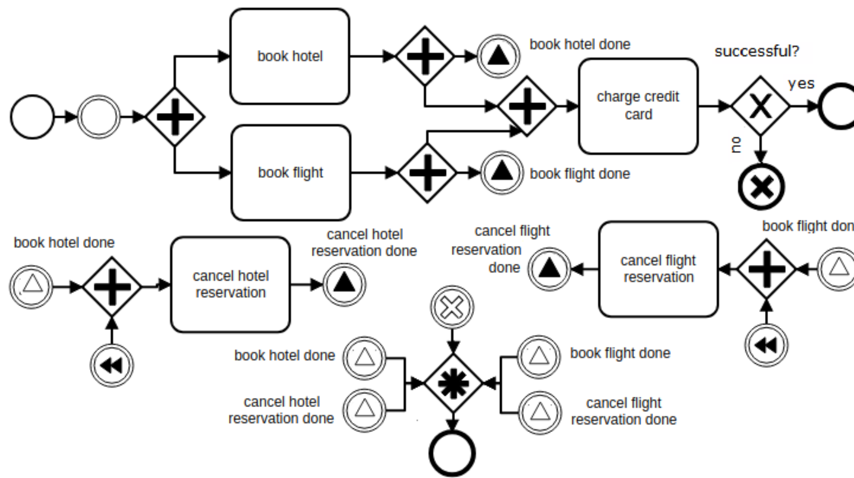
Figure 5.1.1b demonstrates the result of applying the transaction refinement algorithm on a sample transaction shown in Figure 5.1.1a.

Listing 5.1: Refinement of transactions

```
1  refine(BPMN2Process proc) {
2    foreach (Transaction tran in proc.objects.filter(e | e.isTypeOf('
        ↪ Transaction'))) {
3
4      Event[] cancels = tran.objects.filter(e | e.isTypeOf('
          ↪ CatchingCancelEvent'));
5      assert(cancels.length == 1);
6
7      Gateway postCompensation = new ComplexGateway();
8      postCompensation.nextFlowObjects = cancels[0].nextFlowObjects;
9      cancels[0].nextFlowObjects = {postCompensation};
10
11     foreach(Task start : tran.objects.filter(e | e.isTypeOf('Task')
          ↪ ∧ e.compensation != null) and tran.previousFlowObjects().
          ↪ length == 0) {
12
13       // Allow post compensation flow only when all performed
            ↪ compensatable tasks are compensated
14       Event taskDone = new CatchingSignalEvent();
15       getDoneSignal(task).receivers.add(taskDone);
16       taskDone.nextFlowObjects = {postCompensation};
17
18       Event compensationDone = new CatchingSignalEvent();
19       getDoneSignal(task.compensation).receivers.add(
            ↪ compensationDone);
20       compensationDone.nextFlowObjects = {postCompensation};
21     }
22
23     foreach (CompensatableTask task in tran.objects.filter(e | e.
          ↪ isTypeOf('Task') ∧ e.compensation != null)) {
24       handleTaskCompletion(task);
25       handleCompensation(cancels[0], task);
26     }
27   }
28 }
```



(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)

```
1  handleTaskCompletion(CompensatableTask task) {
2    // A send signal event to indicate the task is done
3    Event doneSendEvent = new SendSignalEvent();
4    // A receive signal event to catch the signal above
5    Event doneReceiveEvent = new CatchingSignalEvent();
6    doneSendEvent.receivers = {doneReceiveEvent};
7    // Placing the signal event after the task
8    if (task.nextFlowObjects == null) {
9      task.nextFlowObjects = {doneSendEvent};
10   } else {
11     Gateway gateway = new InclusiveGateway();
12     gateway.nextFlowObjects = task.nextFlowObjects;
13     gateway.nextFlowObjects.add(doneSendEvent);
14     task.nextFlowObjects = {gateway};
15   }
16 }
```

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)

```
1  handleCompensation(CatchingCancelEvent cancel, CompensatableTask
       ↪ task) {
2    Event receiver = getDoneSignal(task).receivers[0];
3    CompensatableTask[] nexts = getNextCompensatables(task);
4    if (nexts.length == 0) {
5      Gateway gateway = new InclusiveGateway();
6      cancel.nextFlowObjects.add(gateway);
7      receiver.nextFlowObjects = {gateway};
8      gateway.nextFlowObjects.add(task.compensation);
9    } else {
10     // A complex gateway that fires if either all or
11     // only the first two of its inputs have flow
12     Gateway order = new ComplexGateway();
13     cancel.nextFlowObjects.add(order);
14     receiver.nextFlowObjects.add(order);
15
16     foreach(CompensatableTask next in nexts) {
17       // Event associated with the next compensatable task
18       getDoneSignal(next).nextFlowObjects.add(order);
19
20       // Event associated with compensation of the next
21       // compensatable task
22       Event compensationDone = getDoneSignal(next.compensation).
             ↪ receivers[0];
23       getDoneSignal(compensationDone).nextFlowObjects.add(order);
24     }
25     order.nextFlowObjects.add(receiver);
26   }
27 }
```

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.

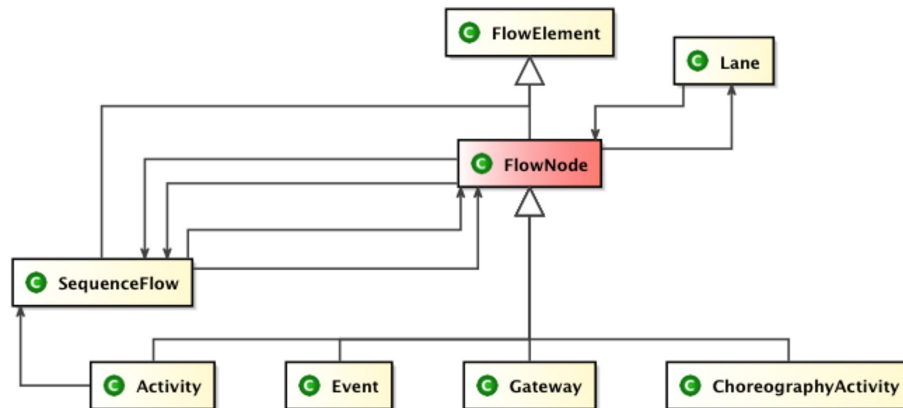


Figure 5.3.1: The FlowNode and its related entities in BPMN 2 EMF meta-model

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

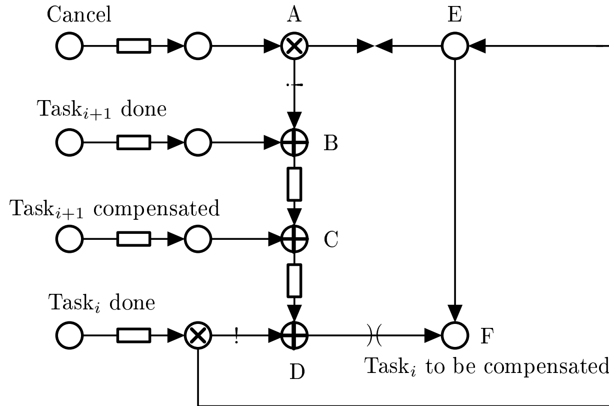


Figure 5.3.2: Mapping of the compensation order complex gateway

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;
```

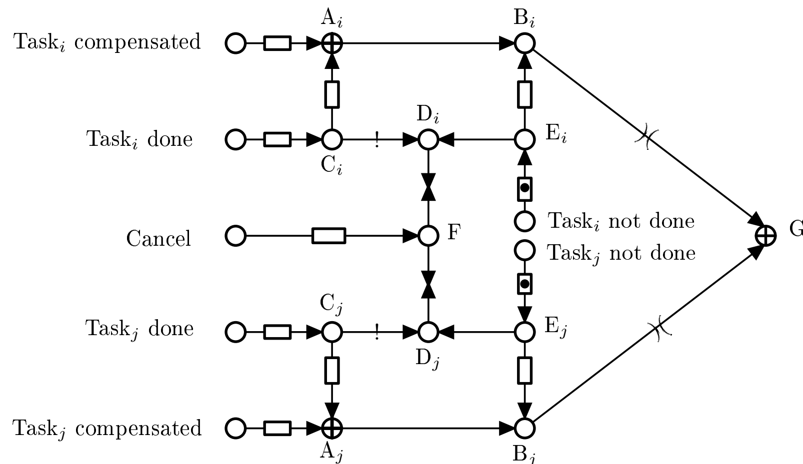


Figure 5.3.3: Mapping of the post compensation complex gateway

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);
         }
      }
}
```

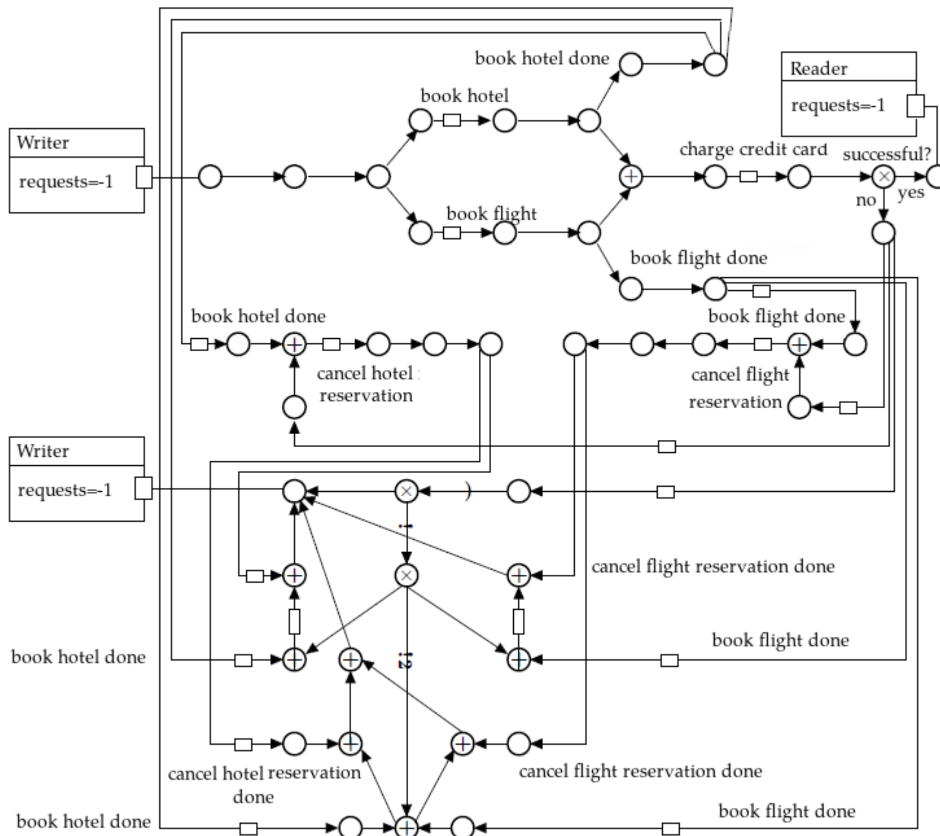


Figure 5.3.4: Mapping the refined BPMN 2 example of Figure 5.1.1b to Reo

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.