

Specification and Implementation of Workflow Control Patterns In Reo

by

Seyedeh Elham Mousavi Bafrooi

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2006

© Seyedeh Elham Mousavi Bafrooi 2006

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Coordination models and languages are relatively new methods in modeling component-based software systems. These models and languages separate the communication aspect of systems from their computation aspect, and hence enable the modeling of concurrent, distributed, and heterogeneous systems. In this thesis, our goal is to show that Reo, a channel-based exogenous coordination language, is powerful enough to be used in the area of workflow management. In order to achieve this goal, we consider a set of workflow control patterns. We implement each of these patterns in terms of a Reo circuit and show that these Reo circuits capture the behavior of the corresponding workflow control patterns. We believe that the patterns we choose in this thesis are enough to show the strength of Reo as a workflow language.

We explain our approach in four steps. In the first step, we specify the general definition of workflow control patterns in terms of some Point Interval Temporal Logic formulas. In the second step, we convert each PITL formula to a constraint automaton. In the third step, we implement each workflow control pattern by a Reo circuit; each Reo circuit consists of a set of *components* and a set of *connectors* that connect and coordinate those components and provide its behavior as a relation on *timed data streams*; a timed data stream is a twin pair of a data stream and a time stream. In the fourth step, we compositionally derive the constraint automata of that Reo circuit and finally, in the fifth step, we show the equivalence of the two constraint automata.

Acknowledgments

I would like to thank my advisor Professor Farhad Mavaddat for his knowledge and constructive criticisms and suggestions throughout this research.

I wish to express my sincere appreciation and gratefulness to my professor Dr. Marjan Sirjani for her insight, valuable guidance, and continuous encouragement throughout my study. Indeed, my gratitude to her can not be expressed in a few words and it is my honor to be her student.

I am thankful to Dr. Farhad Arbab and Dr. Paulo Alencar for their useful comments.

I would also like to take this opportunity to thank the School of Computer Science for supporting me financially and providing me with their facilities.

Last but not the least, a world of thanks goes to my sister Mojgan Daneshmand and my brother Pedram Mousavi who with their open arms welcomed me into their warm home from which I was able to conduct my studies. Their continuous support and kindness is unforgettable.

*To my parents:
For their unconditional love, and endless effort,
I am always grateful to them*

Table of Contents:

Abstract	iii
Acknowledgments	iv
Table of Contents	vi
List of Tables	x
List of Figures	xi
Chapter 1 Introduction	1
1.1. Goal	1
1.2. Approach	1
1.3. Structure of the Thesis	2
Chapter 2 The Reo Coordination Language	3
2.1. Introduction	3
2.2. Basic Concepts	4
2.3. Timed Data Stream	7
2.4. Reo Primitive Channels	8
2.4.1. The <code>sync</code> Channel	9
2.4.2. The <code>filter</code> Channel	10
2.4.3. The <code>lossySync</code> Channel	10
2.4.4. The <code>syncDrain</code> Channel	11
2.4.5. The <code>syncSpout</code> Channel	11
2.4.6. The <code>fifo</code> and <code>fifo₁</code> Channels	12
2.4.7. The <code>asyncDrain</code> Channel	13
2.4.8. The <code>asyncSpout</code> Channel	13
2.4.9. The <code>replicator</code> Connector	14
2.4.10. The <code>merger</code> Connector	15
2.5. Reo Operations	16

2.5.1. The Read Operation.....	16
2.5.2. The Take Operation.....	17
2.5.3. The Write Operation	17
2.6. Reo Patterns.....	17
2.7. Composing Connectors	18
2.8. Constraint Automata: An Operational Model for Reo	19
Chapter 3 Workflow Management Systems	21
3.1. Introduction	21
3.2. Basic Concepts	22
3.2.1. Workflow	23
3.2.2. Workflow Management System.....	23
3.2.3. Business Process, Process Definition, and SubProcess	23
3.2.4. Activity.....	24
3.2.5. Instance (of a Process or an Activity)	24
3.3. Workflow Control Patterns.....	24
Chapter 4 Specification and Implementation of Workflow Patterns	25
4.1. Introduction	25
4.1.1. The Delay Connector.....	28
4.2. Specification and Implementation of Basic Patterns.....	32
4.2.1. Sequential Routing.....	32
4.2.1.1. Workflow Sequential Routing.....	32
4.2.1.2. Reo Sequential Routing	34
4.2.2. AND-Split	35
4.2.2.1. Workflow AND-Split	35
4.2.2.2. Reo AND-Split	36
4.2.3. AND-Join.....	38
4.2.3.1. Workflow AND-Join	38
4.2.3.2. Reo AND-Join	39
4.2.4. XOR-Split	40

4.2.4.1. Workflow XOR-Split.....	41
4.2.4.2. Reo XOR-Split.....	42
4.2.5. XOR-Join	43
4.2.5.1. Workflow XOR-Join.....	44
4.2.5.2. Reo XOR-Join.....	44
4.3. Specification and Implementation of Advanced Synchronization Patterns	46
4.3.1. OR-Split	46
4.3.1.1. Workflow OR-Split.....	46
4.3.1.2. Reo OR-Split.....	48
4.3.2. Synchronizing Merge	49
4.3.2.1. Workflow Synchronizing Merge	50
4.3.2.2. Reo Synchronizing Merge	51
4.3.3. Multi Merge	53
4.3.3.1. Workflow Multi Merge.....	53
4.3.3.2. Reo Multi Merge.....	54
4.3.4. Discriminator.....	55
4.3.4.1. Workflow Discriminator	55
4.3.4.2. Reo Discriminator	57
Chapter 5 Conclusion.....	59
5.1. Summary and Conclusions	59
Appendix A: Other Workflow Control Patterns.....	61
A.1. Specification and Implementation of Structural Patterns	61
A.1.1. Arbitrary Cycles	61
A.1.1.1. Workflow Arbitrary Cycles	61
A.1.1.2. Reo Arbitrary Cycles	62
A.2. Specification and Implementation of Patterns Involving Multiple Instances..	64
A.2.1. Multiple Instances without Synchronization.....	64
A.2.1.1. Workflow Multiple Instances without Synchronization	65
A.2.1.2. Reo Multiple Instances without Synchronization	66
A.2.2. Multiple Instances with Design Time Knowledge	67

A.2.2.1. Workflow Multiple Instances with Design Time Knowledge	68
A.2.2.2. Reo Multiple Instances with Design Time Knowledge	68
A.2.3. Multiple Instances with Run Time Knowledge.....	70
A.2.3.1. Workflow Multiple Instances with Run Time Knowledge.....	70
A.2.3.2. Reo Multiple Instances with Run Time Knowledge.....	71
A.2.4. Multiple Instances without Run Time Knowledge	73
A.2.4.1. Workflow Multiple Instances without Run Time Knowledge	73
A.2.4.2. Reo Multiple Instances without Run Time Knowledge.....	74
A.3. State-based Patterns	77
A.3.1. Deferred Choice	77
A.3.1.1. Workflow Deferred Choice.....	77
A.3.1.2. Reo Deferred Choice.....	78
A.3.2. Interleaved Parallel Routing.....	79
A.3.2.1. Workflow Interleaved Parallel Routing	79
A.3.2.2. Reo Interleaved Parallel Routing	81
A.3.3 Milestone	83
A.3.2.1. Workflow Milestone	83
A.3.2.2. Reo Milestone	85
A.4. Cancellation Patterns	87
A.4.1. Cancel Activity.....	87
A.4.1.1. Workflow Cancel Activity	87
A.4.1.2. Reo Cancel Activity	88
A.4.2. Cancel Case.....	89
A.4.1.1. Workflow Cancel Case	89
A.4.1.2. Reo Cancel Case	90
Appendix B: Overview of Point Interval Temporal Logic.....	91
B.1. Basic Concepts.....	91
References.....	94

List of Tables

Table 2-1 Primitive channels	9
Table 2-2 Reo operations.....	16
Table 2-3 Primitive channels and their constraint automata.....	20

List of Figures

Figure 2-1 Reo circuits.....	5
Figure 2-2 Nodes in Reo.....	6
Figure 2-3 replicator connector.....	14
Figure 2-4 merger connector.....	15
Figure 3-1 The relationship among basic concepts.....	22
Figure 4-1 exclusiveRouter connector.....	29
Figure 4-2 The Delay connector.....	31
Figure 4-3 workflow Sequential Routing pattern.....	33
Figure 4-5 Reo Sequential Routing.....	34
Figure 4-7 workflow AND-Split pattern.....	35
Figure 4-10 Reo AND-Split.....	37
Figure 4-13 workflow AND-Join pattern.....	39
Figure 4-14 Reo AND-Join connector.....	40
Figure 4-15 workflow XOR-Split Pattern.....	41
Figure 4-16 Reo XOR-Split connector.....	42
Figure 4-17 workflow XOR-Join pattern.....	44
Figure 4-18 Reo XOR-Join connector.....	45
Figure 4-19 workflow OR-Split pattern.....	47
Figure 4-20 Reo OR-Split connector.....	48
Figure 4-21 workflow synchronizing merge pattern.....	50
Figure 4-22 Reo Synchronizing Merge connector.....	52
Figure 4-23 workflow Multi Merge pattern.....	53
Figure 4-24 Reo Multi Merge connector.....	55
Figure 4-25 workflow discriminator pattern.....	56
Figure 4-26 Reo Discriminator Connector.....	58
Figure 4-27 n out of m join in Reo.....	58
Figure A-5-1 Workflow Arbitrary Cycles.....	62
Figure A-5-2 Reo Arbitrary Cycle example.....	63
Figure A-5-3 Workflow Multiple Instances without Synchronization.....	65

Figure A-5-4 Reo MI without synchronization connector.....	66
Figure A-5-5 Workflow Multiple Instances with Design Time Knowledge	68
Figure A-5-6 Reo MI with Design Time Knowledge connector	69
Figure A-5-7 Workflow Multiple Instances with Run Time Knowledge.....	71
Figure A-5-8 Reo MI with Run Time Knowledge connector.....	72
Figure A-5-9 Workflow Multiple Instances without Run Time Knowledge.....	74
Figure A-5-10 Reo MI without Run Time Knowledge connector.....	75
Figure A-5-11 Workflow Deferred Choice.....	78
Figure A-5-12 Reo Deferred Choice connector.....	79
Figure A-5-13 Workflow Interleaved Parallel Routing	80
Figure A-5-14 Reo Interleaved Parallel Routing connector	81
Figure A-5-15 Workflow Milestone	84
Figure A-5-16 Reo Milestone connector	85
Figure A-5-17 workflow Cancel Activity.....	88
Figure A-5-18 Reo Cancel Activity connector	89
Figure A-5-19 Workflow Cancel Case	90

Chapter 1

Introduction

1.1. Goal

In this thesis, our goal is to show that the control mechanisms of the Reo coordination language are powerful enough to handle and implement workflow control patterns. To achieve this goal, we consider a set of workflow control patterns. We implement each of these patterns in terms of a Reo circuit and show that these Reo circuits capture the behavior of the corresponding workflow control patterns.

1.2. Approach

In this section we explain our approach to accomplish this goal. For each workflow control pattern, we follow the four following steps.

1. Write a Point Interval Temporal Logic definition of the control pattern,
2. Convert that formula to a constraint automaton,
3. Develop an equivalent Reo circuit for that pattern,
4. Develop the timed data stream model of Reo circuit,

5. Develop the constraint automaton of that Reo circuit, and
6. Finally, show that the two constraint automata are equivalent.

1.3. Structure of the Thesis

The goal and approach of this thesis are outlined earlier in this chapter. In Chapter 2 we introduce Reo, a coordination language for component composition and timed data streams as coalgebraic formalism to describe Reo circuits.

In Chapter 3, we introduce workflow management systems and basic terms used in this context.

In chapter 4, we present basic and advanced workflow control patterns, their specification in PITL and implementation in Reo.

Chapter 5 summarizes and concludes the thesis.

Chapter 2

The Reo Coordination Language

2.1. Introduction

Modern information systems rely increasingly on combining concurrent, distributed, mobile, reconfigurable and heterogeneous components. New models, architectures, languages and verification techniques are necessary to cope with the complexity induced by such systems. Coordination languages have emerged as a successful approach, in that they provide abstractions that cleanly separate behavior from communication, therefore increasing modularity, simplifying reasoning, and ultimately enhancing software development.

Coordination is the process of building programs by gluing together active pieces [4]. Active pieces here can mean processes, objects with threads, agents, or whole applications.

The Reo coordination language is proposed for composition of software components based on the notion of channels [13]. Reo is a channel-based exogenous¹ coordination model that separates the computation part and coordination part of a software system. The Reo coordination language provides the following features as mentioned in [26]:

¹ Exogenous coordination models and languages enable third-party entities to yield coordination control over the interaction behavior of mutually anonymous entities involved in a collaboration from outside of its participants. They provide a basis for the development of effective glue-code languages [10].

- loose coupling among components,
- support for distribution and mobility of components,
- exogenous coordination,
- dynamic reconfigurability, and
- formal semantics based on a coinductive calculus of flow and (alternatively) on constraint automata¹.

In this chapter we explain Reo and timed data streams (TDS) as a coalgebraic formalism to capture the behavior of Reo circuits.

2.2. Basic Concepts

Reo is an exogenous coordination model for orchestrating communication among computational entities in a concurrent and distributed component-based software system [13]. The main building blocks of Reo are a set of primitive channels, a set of components to be coordinated and a set of nodes. We explain each of these next.

Figure 2.1 shows three Reo circuit samples in which boxes are components, straight lines are channels, and small filled circles are nodes. The set of channels enclosed in dashed lines are a set of cooperating channels called connectors. In addition, Reo also provides a set of operations for components to manipulate connector topology and input/output data [13].

¹ Constraint Automata has been introduced in [13].

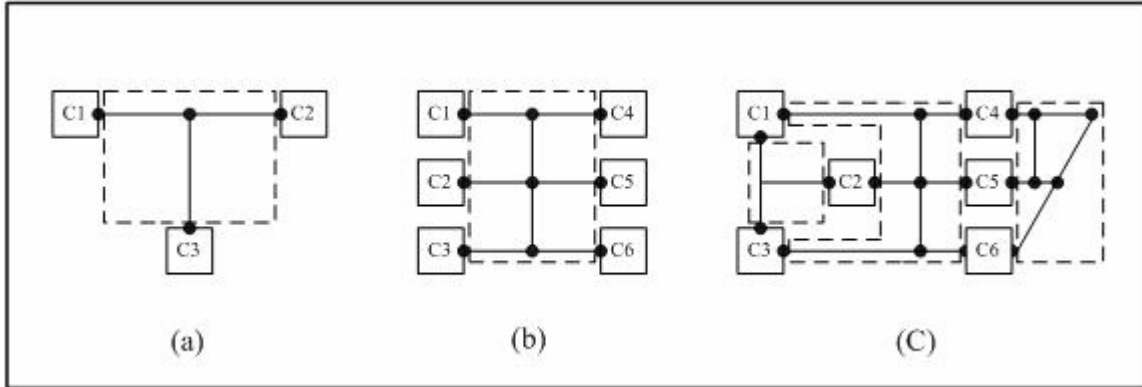


Figure 2-1 Reo circuits; connectors are shown by dashed lines; (a) a 3-way connector; (b) a 6-way connector; (c) two 3-way and one 6-way connectors [13].

Components A component is an abstract type that describes the properties of its instances. A component instance is a set of active entities that can be processes, threads, passive or active objects, a piece of code or even other component instances that are coordinated using Reo connectors. The active entities communicate with the environment using a set of input/output operations provided by Reo. Reo is not aware of the communication and synchronization mechanisms that active entities inside a component instance use to cooperate with each other.

Channels and Connectors Channels are primitive media for communication among component instances. Channels are atomic connectors in Reo. They are dynamically created and automatically garbage collected. Each channel has exactly two ends; *source* end on which data enters the channel and *sink* end from which data leaves the channel. Channel ends can be attached to only one component instance at any time. Reo allows compositional construction of a connector out of primitive channels and other simpler connectors. In subsection 2.3 we introduce primitive channels offered by Reo in more detail.

Nodes A node is a logical construct. In Reo, there are three types of nodes: *source*, *sink* and *mixed*.

- **Source node:** A node is a source node if all channel ends that coincide on it are source channel ends (figure 2.2(c, d)). It acts as a *replicator*; a write

operation to a source node succeeds only if all source channel ends coinciding on this node can accept the data item; thus the data item replicates to all channel ends.

- **Sink node:** A node is a sink node if all channel ends that coincide on it are sink channel ends (figure 2.2(a, b)). A sink node acts as a *merger*¹; a read (or take)² on this node succeeds only if any of the sink channel ends coinciding on this node have a data item ready. If more than one channel end provides data, the sink node will choose one nondeterministically.
- **A mix node:** A node is a mix node if the set of channel ends that coincide on it are a combination of source and sink channel ends (figure 2.2(e)). A mix node nondeterministically chooses one data item from one of the sink channel ends coinciding on it and replicates it to all the source channel ends coinciding on it in an atomic step. ■

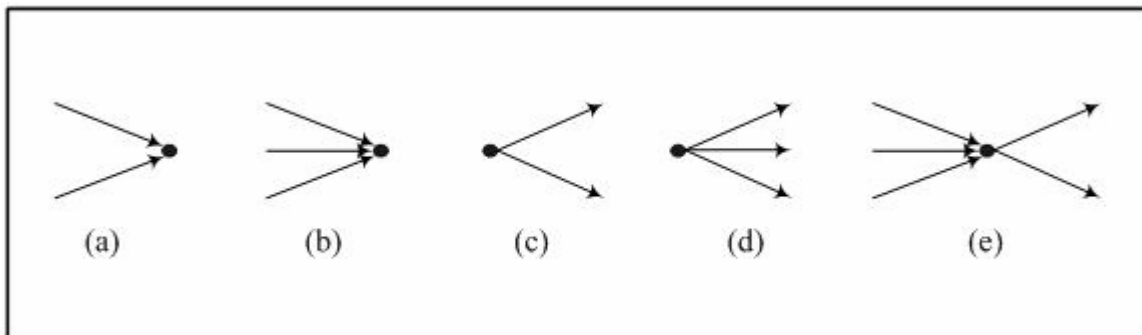


Figure 2-2 Nodes in Reo. Figures (a) and (b) show two source nodes with respectively two and three sink channel ends coinciding on them. Figures (c) and (d) show two sink nodes with respectively two and three source channel ends coinciding on them. Figure (e) shows a mix node with three sink channel ends and two source channel ends coinciding on it [13].

¹ A sink node does not actually merge the data items; it dispenses one data item at a time.

² The read operation is nondestructive which means it copies the data item and does not remove it from a channel. On the other hand, take is destructive which means it removes the data from a channel.

2.3. Timed Data Stream

In Reo, connectors are modeled as relations on *timed data streams*, i.e. timed data streams are coalgebraic formal semantics for Reo connectors [11]. In a timed data stream $\langle \alpha, a \rangle$ the time stream a specifies for each integer $n \geq 0$ the time moment $a(n)$ at which the n th data element $\alpha(n)$ is being input or output.

Let D be an arbitrary set of data items; the set DS is defined as $DS = \{\alpha \mid \alpha: \{0, 1, \dots\} \rightarrow D\}$ and contains data streams $\alpha = (\alpha(0), \alpha(1), \dots, \alpha(n))$. We call $\alpha(0)$ the initial value of α . The derivative α' of the stream α is defined as $\alpha' = (\alpha(1), \dots, \alpha(n))$.

Again, assume \mathbb{R}_+ is the set of non-negative and real numbers, and its elements represent particular time moments; the set TS is defined as $TS = \{a \mid a: \{0, 1, \dots\} \rightarrow \mathbb{R}_+, a < a'\}$ and contains time streams $a = (a(0), a(1), \dots, a(n))$ in which we call $a(0)$ the initial value of a and the derivative a' of the stream a is defined as $a' = (a(1), \dots, a(n))$. With this definition, TS consists of increasing time moments: $a(0) < a(1) < a(2) < \dots$. Moreover, let \leq and $<$ respectively be the regular “strictly smaller” and “smaller” relations on \mathbb{R}_+ , so for any two time streams $a = (a(0), a(1), \dots, a(n))$ and $b = (b(0), b(1), \dots, b(n))$ we have:

$$a < b \equiv \forall n \geq 0, a(n) < b(n), \quad a \leq b \equiv \forall n \geq 0, a(n) \leq b(n)$$

The set TDS of timed data streams is defined as $TDS = DS \times TS$ which contains pairs $\langle \alpha, a \rangle$ consisting of a data stream $\alpha = (\alpha(0), \alpha(1), \dots, \alpha(n))$ in DS and a time stream $a = (a(0), a(1), \dots, a(n))$ in TS .

In next section, we introduce Reo primitive channels and represent their behavior as relations on timed data streams.

2.4. Reo Primitive Channels

Reo supports a collection of predefined channel types, each with its own well-defined behavior. The behavior of a channel, among other parameters, may depend on

- synchronization properties,
- its source and sink ends,
- the size of its buffer,
- its ordering scheme, and
- its loss policy [13].

Generally, Reo channel can be either *synchronous* or *asynchronous*. Besides, they may be *lossy* or not. In a synchronous channel the two operations pending on each end of the channel succeed simultaneously. In an asynchronous channel, data items may be buffered in a (un)bounded buffer and the channel may or may not impose any order on those data items. In a lossy channel, the data item may be delivered, from source to sink end, synchronously, or lost by the channel.

Table 2.1 shows the graphical representations of eight primitive channels provided by Reo. Next, we explain each of those eight channels.



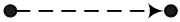

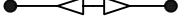



Channel Type	Graphical Representation
sync	
Filter(<i>pat</i>)	
lossySync	
syncDrain	
syncSpout(<i>pat</i>)	
fifo	
asyncDrain	
asyncSpout(<i>pat</i>)	

Table 2-1 Primitive channels

2.4.1. The sync Channel

A `sync` channel represents the typical synchronous channels. A `read` (or `take`) operation on the sink end of this channel succeeds only if there is a `write` operation pending on the source end of this channel; moreover, the type of data item offered by the `write` operation should match that of the `read` (or `take`) operation. A `write` operation, on the other hand, succeeds only if there is a `take` operation pending on the sink end of the channel and the data types of both operations match. Note that the `take` operation is destructive, as opposed to the `read` operation, which is non-destructive.

As a TDS relation, the `sync` channel is defined, for timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$, by

$$\langle \alpha, a \rangle \text{ sync } \langle \beta, b \rangle \equiv \alpha = \beta \wedge a = b$$

This channel inputs the data stream α at times a , and outputs the data stream β at times b so that $\alpha = \beta$ and $a = b$.

2.4.2. The `filter` Channel

A `filter` channel with the signature `filter(pat)` is a lossy synchronous channel that has a special pattern, `pat`, for data items it can accept. If a data item matches with the pattern `pat`, the channel behaves in the same way as `sync` channel; otherwise, corresponding `write` operation succeeds and the data item will be lost and the `read` (or `take`) operation remains pending.

As a TDS relation, the `filter(pat)` channel is defined, for timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$, by

$$\langle \alpha, a \rangle \text{filter}(\text{pat}) \langle \beta, b \rangle \equiv \begin{cases} \alpha(0) = \beta(0) \wedge a(0) = b(0) \wedge \\ \langle \alpha', a' \rangle \text{filter}(\text{pat}) \langle \beta', b' \rangle & \text{if } \text{pat} \\ \langle \alpha', a' \rangle \text{filter}(\text{pat}) \langle \beta, b \rangle & \text{if } \neg \text{pat} \end{cases}$$

It inputs the data stream α at times a , and outputs the data stream β at times b so that if pattern of α matches `pat`, then $\alpha = \beta$; if not, input data element α is simply discarded and its respective `write` operation succeeds. The it continues with the remainder of the streams.

2.4.3. The `lossySync` Channel

A `lossySync` channel is a lossy synchronous channel in which a `write` operation on its source end always succeeds. If there is a `read` (or `take`) operation pending on the sink node of this channel and their data types match, that data will be removed from the channel and both `write` and `read` (or `take`) operations succeed immediately, like a normal synchronous channel; otherwise the `write` operation succeeds and the data item will be lost.

As a TDS relation, the `lossySync` channel is defined, for timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$, by

$$\langle \alpha, a \rangle \text{lossySync} \langle \beta, b \rangle \equiv \begin{cases} \alpha(0) = \beta(0) \wedge \langle \alpha', a' \rangle \text{lossySync} \langle \beta', b' \rangle & \text{if } a(0) = b(0) \\ \langle \alpha', a' \rangle \text{lossySync} \langle \beta, b \rangle & \text{if } a(0) < b(0) \end{cases}$$

It inputs the data stream α at times a , and outputs the data stream β at times b so that if $a(0) = b(0)$, then $\alpha(0) = \beta(0)$; after that it continues with the remainder of the streams as before. If $a(0) < b(0)$, then input stream $\alpha(0)$ is simply discarded and the channel proceed with $\langle \alpha', a' \rangle$ on its input and $\langle \beta, b \rangle$ on its output end .

2.4.4. The `syncDrain` Channel

A `syncDrain` is a lossy synchronous channel that has two source ends. Two `write` operations pending on its source ends succeed only simultaneously and all written values are lost. This channel synchronizes the two `write` operations on its two source ends.

As a TDS relation, the `syncDrain` channel is defined, for timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$, by

$$\langle \alpha, a \rangle \text{syncDrain} \langle \beta, b \rangle \equiv a = b$$

The corresponding data elements in the streams α and β enter the two input ends of this channel simultaneously, $a = b$, and then disappear.

2.4.5. The `syncSpout` Channel

A `syncSpout` channel with the signature `syncSpout` is an unbounded source of data items that can be taken from its two sink ends simultaneously and

nondeterministically. The data items taken by the two `read` (or `take`) operations pending on its sink ends are independent of each other, although the operations are synchronized by the channel.

As a TDS relation, the `syncSpout` channel is defined, for timed data streams $\langle\alpha, a\rangle$ and $\langle\beta, b\rangle$, by

$$\langle\alpha, a\rangle \text{syncSpout} \langle\beta, b\rangle \equiv \alpha = d_1 \wedge \beta = d_2 \wedge a = b$$

Where d_1 and d_2 are independent data streams that copied into streams α and β , respectively and simultaneously, $a = b$.

2.4.6. The `fifo` and `fifo1` Channels

A `fifo` channel is an unbounded asynchronous channel with one source and one sink end. A `write` operation on a `fifo` channel always succeeds, since the buffer is unbounded. A `read` (or `take`) operation on it remains pending until the first item in the buffer matches with the `read` (or `take`) operation pattern. A `fifo1` channel is a bounded asynchronous channel. The suffix 1 represents the capacity of the buffer (we can also have `fifon` channel, in which n is an integer greater than zero). A `write` operation in a `fifo1` channel succeeds only if the buffer is empty.

As a TDS relation, the `fifo1` channel is defined, for timed data streams $\langle\alpha, a\rangle$ and $\langle\beta, b\rangle$, by

$$\langle\alpha, a\rangle \text{fifo}_1 \langle\beta, b\rangle \equiv \alpha = \beta \wedge a < b < a'$$

This is a 1-bounded first-in-first-out buffer. It inputs the data stream α at times a , and outputs the data stream β at times b so that $\alpha = \beta$ and $a < b$. Moreover, at any moment the next data item can be input only after the present data item has been output: $b < a'$.

2.4.7. The `asyncDrain` Channel

An `asyncDrain` channel is a lossy asynchronous channel with two source ends. The two `writes` on its two ends could never succeed simultaneously. The data item of the two ends is lost after the `write` succeeds.

As a TDS relation, the `asyncDrain` channel is defined, for timed data streams $\langle\alpha, a\rangle$ and $\langle\beta, b\rangle$, by

$$\langle\alpha, a\rangle \text{ asyncDrain } \langle\beta, b\rangle \equiv a(0) \neq b(0) \wedge \begin{cases} \langle\alpha', a'\rangle \text{ asyncDrain } \langle\beta, b\rangle & \text{if } a(0) < b(0) \\ \langle\alpha, a\rangle \text{ asyncDrain } \langle\beta', b'\rangle & \text{if } b(0) < a(0) \end{cases}$$

The corresponding data elements in the streams α and β never enter the two input ends of this channel simultaneously, $a(0) \neq b(0)$ and when they enter they disappear; it handles the remainder of the streams in the same manner.

2.4.8. The `asyncSpout` Channel

An `asyncSpout` channel with the signature `asyncSpout(pat)` is an unlimited source of data items. It is an asynchronous channel with two sink ends. The two `read` (or `take`) operations could never succeed simultaneously; they only succeed one at a time when the data item matches the pattern of the `read` (or `take`). The data items are not related to each other.

As a TDS relation, the `asyncSpout` channel is defined, for timed data streams $\langle\alpha, a\rangle$ and $\langle\beta, b\rangle$, by

$$\langle\alpha, a\rangle \text{ asyncSpout } \langle\beta, b\rangle \equiv a(0) \neq b(0) \wedge \begin{cases} \alpha(0) = d_1 \wedge \langle\alpha', a'\rangle \text{ asyncSpout } \langle\beta, b\rangle & \text{if } a(0) < b(0) \\ \beta(0) = d_2 \wedge \langle\alpha, a\rangle \text{ asyncSpout } \langle\beta', b'\rangle & \text{if } b(0) < a(0) \end{cases}$$

Where d_1 and d_2 are independent data elements that copied into streams α and β , respectively but never simultaneously, i.e. $a(0) \neq b(0)$, then it handles the remainder of the streams in the same manner.

Now, we introduce two useful connectors in Reo, `replicator` and `merger` connectors and show their behavior as relations on timed data streams.

2.4.9. The `replicator` Connector

The `replicator` connector is shown in figure 2-3. A write on node A of `replicator` succeeds only if both channels \mathbf{AB} and \mathbf{AC} are capable of consuming a copy of the written data. However, if even one is not prepared to consume the data, the write suspends.

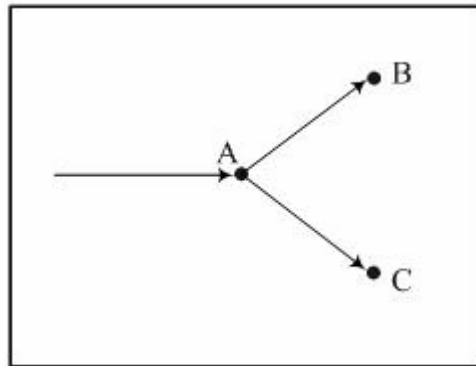


Figure 2-3 `replicator` connector

This connector is a ternary relation with one input end and two output ends. As a TDS relation, the `replicator` connector is defined, for all timed data streams $\langle \alpha, a \rangle$, $\langle \beta, b \rangle$, and $\langle \gamma, c \rangle$, by

$$\text{replicator}(\langle \alpha, a \rangle, \langle \beta, b \rangle, \langle \gamma, c \rangle) \equiv \alpha = \beta = \gamma \wedge a = b = c$$

This connector replicates the data stream α at times a to data streams β and γ at times b and c , respectively, so that $\alpha = \beta = \gamma$ and $a = b = c$.

2.4.10. The merger Connector

The merger connector is shown in figure 2-4. A read (or take) from node C delivers a data value out of channels AC or BC . The data is chosen nondeterministically if both channels have data to write. Thus, the merger connector produces a nondeterministic merge of the values that arrive on A and B .

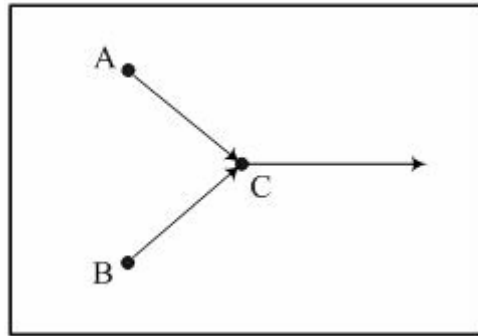


Figure 2-4 merger connector

The connector merge is a ternary relation with two input and one output end, and is defined, for timed data stream $\langle \alpha, a \rangle$, $\langle \beta, b \rangle$, and $\langle \gamma, c \rangle$, by

$$\begin{aligned} \text{merger}(\langle \alpha, a \rangle, \langle \beta, b \rangle, \langle \gamma, c \rangle) \equiv \\ a(0) \neq b(0) \wedge \\ \begin{cases} \alpha(0) = \gamma(0) \wedge a(0) = c(0) \wedge \text{merger}(\langle \alpha', a' \rangle, \langle \beta, b \rangle, \langle \gamma', c' \rangle) & \text{if } a(0) < b(0) \\ \beta(0) = \gamma(0) \wedge b(0) = c(0) \wedge \text{merger}(\langle \alpha, a \rangle, \langle \beta', b' \rangle, \langle \gamma', c' \rangle) & \text{if } b(0) < a(0) \end{cases} \end{aligned}$$

This connector merges the two data streams α and β on its input end into a stream γ on its output end. The data that is handled first is the first one arrives (whether $a(0) < b(0)$ or $b(0) < a(0)$) never at the same time ($a(0) \neq b(0)$). After that, the connector handles the remainder of the streams in the same manner again.

One can use any number of above channels and builds new connectors out of them. It is also possible to define new primitive channels (for more explanation you may see [13]).

2.5. Reo Operations

Reo provides a set of operations to be used by active entities inside a component instance. Table 2-2 shows the operations provided by Reo. Here we introduce three operations, `read`, `take`, and `write` that we will use in our implementations.

Channel Name	Description
<code>create</code>	Creates a new channel and returns the identifiers of two ends.
<code>forget</code>	Changes the value of a channel end so that it does not refer to that channel end anymore.
<code>move</code>	Moves a channel end to a new location.
<code>connect</code>	Connects a channel end to the component instance performing this operation.
<code>disconnect</code>	Disconnects a channel end from the component instance performing this operation.
<code>wait</code>	Succeeds when its related boolean condition becomes true.
<code>join</code>	Joins two channel ends.
<code>split</code>	Splits the node attached to the channel end perform this operation into two new nodes.
<code>hide</code>	Hides the node attached to the channel end perform this operation so that it cannot be modified by any operation.
<code>read</code>	Copies the data item from the channel end without removing it from the channel end.
<code>take</code>	Copies the data item from the sink channel end and removes it from the channel end.
<code>write</code>	Writes a data item to a source channel end if the channel end can accept it.

Table 2-2 Reo operations.

2.5.1. The `read` Operation

The `read` operation, with signature `read([t,]inp[,v[,pat]])`, succeeds if the sink channel end `inp` has a data item ready that matches the pattern `pat`; if so, the data item will be copied to variable `v`. The `read` operation is not destructive, so the data

item will not be removed from the channel end. The optional parameter t indicates a time out value greater than or equal to zero. If the operation does not succeed within the specified time, a failure notification is returned.

2.5.2. The `take` Operation

The `take` operation, with signature `take([t,]inp[, v[, pat]])`, succeeds if the sink channel end `inp` has a data item ready that matches the pattern `pat`; if so, the data item will be copied to variable `v`. The `take` operation is destructive, so the data item will be removed from the channel end. The optional parameter t indicates a time out value greater than or equal to zero. If the operation does not succeed within the specified time, a failure notification is returned.

2.5.3. The `write` Operation

The `write` operation, with signature `write([t,]outp, v)`, succeeds if the source channel end `outp`, can accept the data item offered by the `write` operation; if so the content of variable `v` will be consumed by channel end `e`. The optional parameter t indicates a time out value greater than or equal to zero. If the operation does not succeed within the specified time, a failure notification is returned. ■

In implementing workflow patterns with Reo we assume that activities perform only `take` operation, so that the respective `write` operation can always succeed.

2.6. Reo Patterns¹

Reo uses patterns to regulate channel input/output operations [13]. A pattern is an expression that matches a data item when it flows through a channel. The operations `take` and `read` can specify patterns that must match the items they read. Moreover,

¹ We use the term *pattern* in two different context throughout this thesis; one in the context of Reo operations and the other in the context of workflow management. We will clearly distinguish these two usages of pattern by explicitly indicating Reo operation patterns and workflow control patterns.

some channel types may require patterns as their creation parameters to influence their behavior (e.g. `filter(pat)` and `syncSpout(pat)` channels).

Atomic patterns identify types such as *int*, *real*, *string*, *number*, etc.; they match with any one of their instances. Patterns can be composed into tuple structures using "<" and ">". Moreover, a pattern can be augmented with additional conditions in square brackets. For instance, the pattern `<int x, string [a + b * c], int y > [y ≥ 3 * x]` matches triplets consisting of two integers, *x* and *y*, a string, where the second argument is greater than or equal to 3 times the first one, and the string consists of one or more occurrences of "a" followed by zero or more occurrences of "b" with a single "c" at its end.

2.7. Composing Connectors

Since connectors are relations, their composition can be modeled by relational composition [10]. We present an example to illustrate how connectors are composed. The following relations show the composition of two `sync` channels.

$$\begin{aligned}
 \langle \alpha, a \rangle \text{sync} \langle \gamma, c \rangle \circ \langle \gamma, c \rangle \text{sync} \langle \beta, b \rangle &\equiv \langle \alpha, a \rangle \text{sync} \langle \gamma, c \rangle \wedge \langle \gamma, c \rangle \text{sync} \langle \beta, b \rangle \\
 &\equiv (\alpha = \gamma \wedge a = c) \wedge (\gamma = \beta \wedge b = c) \\
 &\equiv \alpha = \beta \wedge a = b \equiv \langle \alpha, a \rangle \text{sync} \langle \beta, b \rangle
 \end{aligned}$$

The composition is obtained by simply applying logical rules to TDS formulas. The general definition of the composition of an arbitrary *n*-ary connector *R* and *m*-ary connector *T*, is essentially the same. One has to identify and connect an input end from *R* to an output end from *T*. It is also possible to connect a single output of *R* to several inputs ends from *T* at the same time, or reversely, connect several input ends from *T* to a single output end from *R*. The results of the last two compositions are `replicator` and `merger` connectors, respectively, which were introduced in the previous section.

2.8. Constraint Automata: An Operational Model for Reo

Constraint automata are an operational model for Reo. It is a formalism to describe the behavior and possible data flow in Reo circuits (coordination models in general) and provides a basis for formal verification of those circuits. In a constraint automaton of any Reo circuit, the automata states represent the possible configurations and the automata transitions represent the possible data flow and the effect of them on configurations. Table 2-3 shows the constraint automata for Reo primitive channels. Constraint automata also serve as acceptors for timed data streams or TDS-languages. It observes the data at some input/output ports of components and either changes its state or rejects the data if there is no corresponding transition to be fired by that data. So, constraint automata are also a formalism to describe TDS-languages.

Constraint automata use a finite set N of *names*, e.g., $N = \{A_1, \dots, A_n\}$ where A_i stands for the i -th input/output port of a connector or component. The transitions of constraint automata are labeled with pairs consisting of a non-empty subset N of $\{A_1, \dots, A_n\}$ and a data constraint g . Data constraints can be viewed as a symbolic representation of sets of data-assignments. Formally, data constraints are propositional formulae built from the atoms " $dA = d$ " where data item d is assigned to port A . Data constraints are given by the following grammar: $g ::= true \mid d_A = d \mid g_1 \vee g_2 \mid \neg g$.

Definition 1: A constraint automaton (over the data domain $Data$) is a tuple $A = (Q, Names, \rightarrow, Q_0)$ where

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

We call N the name-set and g the guard of the transition. For every transition $q \xrightarrow{N,g} p$ we require that: (1) $N \neq \emptyset$, and (2) $g \in DC(N, Data)$. A is called finite iff Q , \rightarrow and the underlying data domain $Data$ are finite. In a similar way, we may define the natural join for TDS-languages with other name-sets. Thus, join as an operator for TDS-languages can be regarded as a generalization of intersection. It is realized on the automata-level by a product-construction.

Definition 2: The product-automaton of the two constraint automata $A_1 = (Q_1, Names_1, \rightarrow_1, Q_{0,1})$ and $A_2 = (Q_2, Names_2, \rightarrow_2, Q_{0,2})$, is:

$$A_1 \circ A_2 = (Q_1 \times Q_2, Names_1 \cup Names_2, \rightarrow, Q_{0,1} \times Q_{0,2})$$

where \rightarrow is defined by the following rules:

$$\frac{q_1 \xrightarrow{N_1, g_1} p_1, \quad q_2 \xrightarrow{N_2, g_2} p_2, \quad N_1 \cap Names_2 = N_2 \cap Names_1}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle}$$

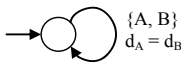
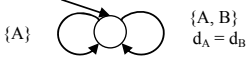


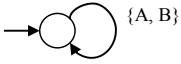
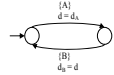
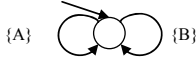

Channel Type	Constraint Automata
sync	 $\{A, B\}$ $d_A = d_B$
Filter(pat)	 $\{A\}$ $\{A, B\}$ $d_A = d_B$
lossySync	 $\{A\}$ $\{A, B\}$ $d_A = d_B$
syncDrain	 $\{A, B\}$
syncSpout(pat)	 $\{A, B\}$
fifo $_n$	 $\{A\}$ $\{B\}$ $d_B = d$
asyncDrain	 $\{A\}$ $\{B\}$
asyncSpout(pat)	 $\{A\}$ $\{B\}$

Table 2-3 Primitive channels and their constraint automata

Chapter 3

Workflow Management Systems

3.1. Introduction

Workflow Management Systems are used to automate business processes. They control the flow of work through a company, thus providing the right person the right task at the right point of time. This helps in streamlining the business processes for a company and has the potential to improve the productivity of recurring tasks by a significant amount. The introduction of workflow management systems results in a gain in efficiency and productivity, increased security, clear progress reports as well as quality and cost control benefits.

Workflow management deals with supporting business processes in organizations; it involves managing the flow of work through an organization [12]. Workflows are a collection of coordinated tasks designed to carry out a well-defined complex process [11]. A workflow management system is a generic information system that supports modeling, execution, as well as the management and monitoring of workflows. Such a system operates on a workflow specification, a description of the business processes in the organization that should be supported.

Workflow modeling is the task of creating workflow specifications. Such specifications will usually be used as input to a workflow management system. A set of workflow control patterns has been introduced in [37]. In this thesis, we consider those patterns. In the following sections, we define basic concepts of workflow management

systems that we use throughout this thesis and describe the workflow control patterns as categories by [37].

3.2. Basic Concepts

In this section, we define some of the terms in the context of workflow management that we will use in this thesis. These definitions are mostly from *WfMC Coalition, Terminology and Glossary* [39]. Figure 3-1 shows the relationship between basic concepts we define in this section¹.

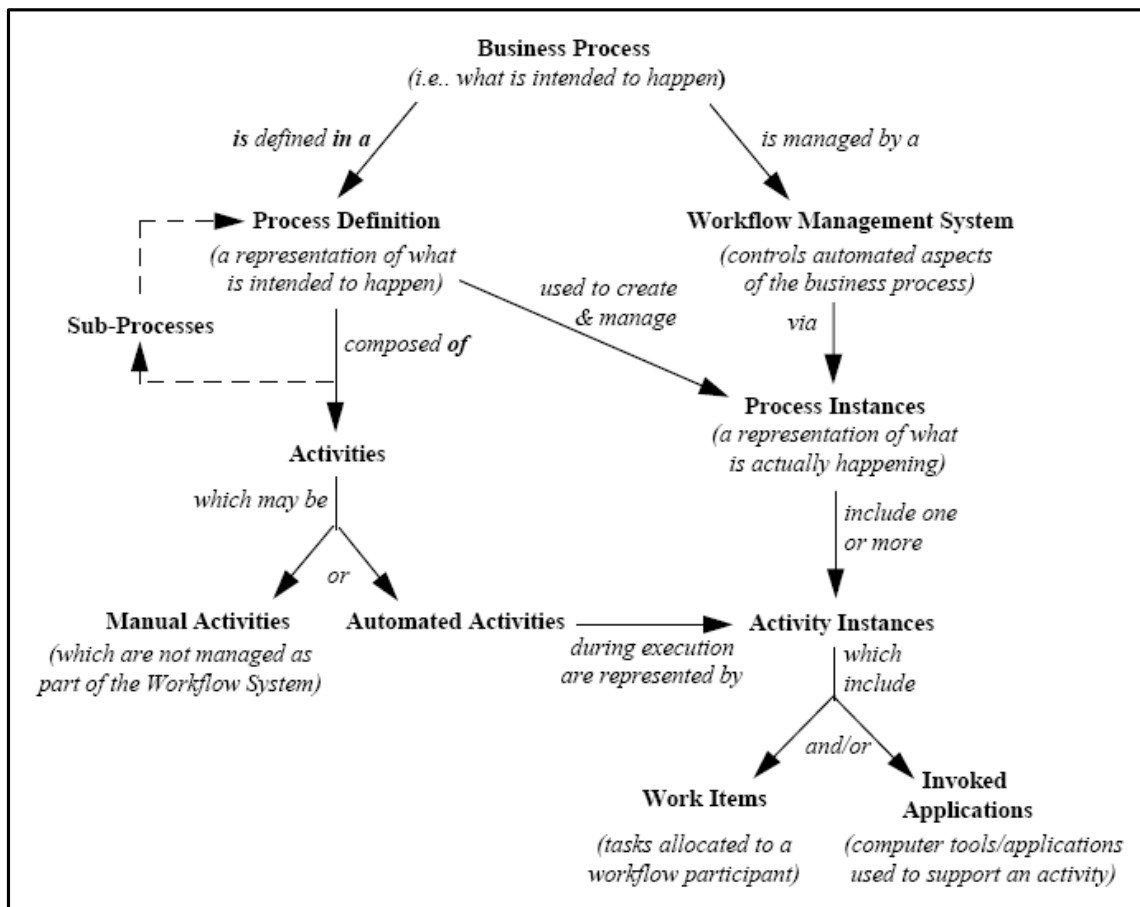


Figure 3-1 The relationship among basic concepts [39].

¹ All the explanations given in this section are not necessary for understanding this thesis; they are given for the sake of completeness.

3.2.1. Workflow

Workflow is the automatic routing of business processes. Responsible participants and users receive their desired documents and information and perform some form of processing on the documents based on procedural rules. The documents may be physically moved over the network or maintained in a single database with the appropriate users given access to the data at the required times.

3.2.2. Workflow Management System

Workflow management deals with modeling and controlling the execution of application processes in an organizational or technical environment [3]. A workflow management system is a software system working on a workflow engine and is responsible for storing the process definitions, creating and executing workflow instances (i.e. automated parts of business processes), managing the interaction among workflow instances and participants, among other applications. These systems may also provide some administrative functions such as work reassignment.

3.2.3. Business Process, Process Definition, and SubProcess

Business processes are sets of activities involved within or outside an organization that work together to produce a business outcome for an organization. A business process is typically associated with operational objectives and business relationships. It may consist of automated activities, capable of workflow management, and/or manual activities, which lie outside the scope of workflow management.

The process definition consists of a network of activities and their relationships, criteria to indicate the start and termination of the process, and information about the individual activities, such as participants, associated IT applications and data, etc. It is defined to specify which activities need to be executed and in what order. The process definitions are instantiated for different cases. A subprocess is useful for defining reusable components within other processes. It has its own process definition, and may

include parameters passed on its initiation and completion. Multiple levels of subprocess may be supported.

3.2.4. Activity

An activity is an atomic (and the smallest) piece of work (i.e. executed in one logical step). They are connected together using transitions. Activities may be automated (workflow activity) or manual; the automated activity is one that can be computerized and the manual activity is the one that does not support automation and relies outside the scope of workflow management. Activities require human and/or computer resources in order to be executed.

3.2.5. Instance (of a Process or an Activity)

Process instance is the representation of a single enactment of a process. Activity instance is the representation of an activity within a process instance. Both are created and managed by a workflow management system for each separate invocation of the process or activity. Each instance represents a separate thread of execution of the process or activity, which may be controlled independently and will have its own internal state. Both process instance and activity instance are created, managed and (eventually) terminated by a workflow management system, in accordance with the process definition.

3.3. Workflow Control Patterns

The set of control flow patterns that we use in this thesis is the one introduced by *van der Aalst et al.* [37], based on the various features available in existing systems and common recurring business requirements. The “control flow patterns” are divided in several groups: basic patterns, advanced branching patterns, structural patterns, multiple instance patterns, state-based patterns, and cancellation patterns. In chapter 4 and Appendix A we introduce some of the patterns with their specification in PITL and their implementation in Reo.

Chapter 4

Specification and Implementation of Workflow Patterns

4.1. Introduction

In this chapter, we present the specification and implementation of workflow basic and advanced synchronization patterns. Specification and implementation of other patterns, i.e. structural patterns, patterns with multiple instances, state based patterns, and cancellation patterns are presented in Appendix A.

In this chapter, for each workflow pattern, we first provide the specification of the general concept of that pattern in a point interval temporal logic (PITL) formula. The explanation of the point interval temporal logic is presented in Appendix B. Then, we present the implementation of the pattern in Reo. Next, for each implemented circuit, we present the behaviour of that Reo circuit as a timed data stream relation.

To show that the specification of a pattern in PITL and its implementation in Reo are equal, our approach is as follow. For one pattern, we convert the specification of that pattern in PITL to a constraint automaton. Then, we provide the constraint automaton for the Reo circuit that implements that pattern and finally show that these two constraint automata are equal. We follow this approach for two patterns, Sequential Routing and AND-Split, for the rest of the patterns the approach is the same and we do not provide it in this thesis.

Each workflow case consists of several paths to be taken and each path has one or more activities that will be executed in some orders. Obviously, if two paths have the same number of activities with a particular ordering, those paths are identical. Moreover, if two workflow cases have identical paths those workflow cases are equal in terms of showing the same behavior. In this thesis, we show paths as a set of activities with some ordering denoted by time and show that these paths are identical so the corresponding workflow cases are identical as well. Next we explain why we have chosen the notion of time for the specifications and correctness of the implementations.

What a particular control pattern is trying to do, given a workflow case consisting of several activities, is to control the execution order and synchronization of those activities. By execution order we mean that the control pattern decides which activities should become enabled after the completion of what other activities and in what order. The control pattern is not concerned with how an enabled activity will be executed and whether it works as it is supposed to or not. The control flow pattern considers each activity as a black box and the only observable behavior which the pattern is concerned about is *when* the activity becomes enabled and *when* it finishes executing. Or more accurately, what should happen *after* an activity finishes its execution and what should have happened *before* an activity becomes enabled. Obviously, the words *before* and *after* both represent time semantics.

By synchronization we mean that after completion of an activity the control pattern decides whether the execution thread should split into multiple threads and whether all of these threads should be executed in parallel or they are required not to be executed in parallel. Furthermore, it determines how these multiple threads should be merged; whether all of them should be synchronized so that the next activity can become enabled or it is sufficient that certain number of them completes and be synchronized. Again, it is obvious that synchronization implicitly poses some timing order on the execution of the activities in a workflow case.

Here are some definitions in the context of specification of workflow patterns with PITL and in the context of implementing patterns in Reo and their formalization in TDS.

1. In the contest of specifying workflow patterns with PITL, let A be an activity. We refer to the interval during which activity A is enabled and being executed by A . This interval is defined as $A = [A_s, A_e]$, in which A_s is the start point of interval A ; it is the moment at which activity A becomes enabled. A_e is the end of interval A ; it is the moment at which activity A finishes executing. Moreover, we assume that always $A_s < A_e$, which means that interval A is a non-zero length interval (see Appendix B for definition of zero and non-zero length intervals).

2. In the contest of implementing workflow patterns with Reo, for each activity A we have:
 - $\langle d(A_I), \tau(A_I) \rangle$: Let A_I be the input node of activity A . This tuple means that A performs a `take` operation on A_I and takes data stream $d(A_I)$; thus, $\tau(A_I)$ is the moment at which activity A becomes enabled.

 - $\langle d(A_O), \tau(A_O) \rangle$: Let A_O be the output node of activity A . This tuple means that A performs a `write` operation on A_O and writes data stream $d(A_I)$; thus, $\tau(A_O)$ is the moment at which activity A finishes executing.

 - Furthermore, we assume that $\tau(A_I) < \tau(A_O)$.

From definitions 1 and 2 it is obvious that for each activity A ,

$$\tau(A_I) = A_s \quad \tau(A_O) = A_e \quad (4-1)$$

Consider the TDS formula below which represents a `sync` channel.

$$\langle d(A), \tau(A) \rangle \text{sync} \langle d(B), \tau(B) \rangle \equiv d(A) = d(B) \wedge \tau(A) = \tau(B)$$

Recall from section 2.3 that the $d(A)=d(B)$ statement indicates that the data written on node A is taken on node B required that the type of data on node A matches the expected type at node B . If we assume that the `take` operation does not restrict the type of data it takes, then it can accept any type of data; this way the $d(A)=d(B)$ statement always is true, which means that eventually writing and taking the data from the source and sink nodes always happens successfully.

Besides, recall that in the context of modeling workflow control patterns we are only interested in the execution ordering of activities not on the data they pass to each other to perform their tasks.

Considering the above explanations, the $d(A)=d(B)$ statement does not have any impact on modeling workflow control patterns. Hence, when we want to show that the specification and the implementation of patterns are identical, we only consider the time sequence part of the TDS formula (e.g. $\tau(A)=\tau(B)$). But in specifying the behaviour of connectors that implement workflow patterns we represent the complete TDS formula.

Next, we introduce a Reo connector that we are going to use to implement our circuits with.

4.1.1. The Delay Connector

In this section, we introduce a new connector, `Delay`, which we use to connect and coordinate the activities in this chapter and Appendix A. In Reo, as we mentioned in chapter 2, there are three sets of channels: synchronous, asynchronous, and lossy channels. If we use only synchronous channels to connect the activities, finishing of one activity and enabling of the next activity always happen simultaneously. If we use asynchronous channels, on the other hand, finishing of one activity and enabling of the next activity always happen with a delay. None of these cases is desired alone in the context of workflow. In a workflow case, finishing of one activity and enabling of the

next activity can happen either simultaneously or with some delay. So, we must define a new connector that allows both behaviours.

We use the `exclusiveRouter` connector to build the `Delay` connector. The detailed explanation of `exclusiveRouter` is presented in [10]. Figure 4.1(a) shows the structure of this connector. In this connector, when a data arrives at node a , it flows through the circuit and is finally written on either node b or node c , but not both. We use the notation in figure 4.1(b) to represent the `exclusiveRouter` connector. In the following paragraph we compositionally derive the behaviour of this connector as a timed data streams relation.

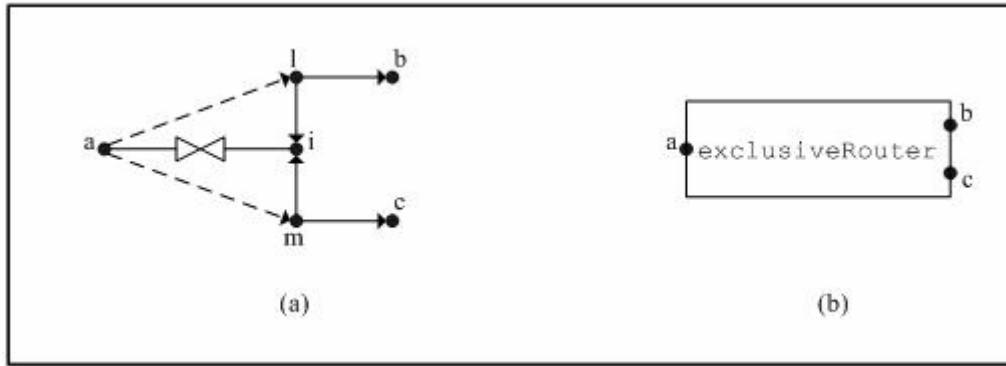


Figure 4-1 `exclusiveRouter` connector

1. For the `lossySync` channel between nodes a and l we have:

$$\langle d(a), \tau(a) \rangle \text{lossySync} \langle d(l), \tau(l) \rangle \equiv \begin{cases} d(a_0) = d(l_0) \wedge \langle d(a)', \tau(a)' \rangle \text{lossySync} \langle d(l)', \tau(l)' \rangle & \text{if } \tau(a_0) = \tau(l_0) \\ \langle d(a)', \tau(a)' \rangle \text{lossySync} \langle d(l), \tau(l) \rangle & \text{if } \tau(a_0) < \tau(l_0) \end{cases}$$

2. For the `lossySync` channel between nodes a and m we have:

$$\langle d(a), \tau(a) \rangle \text{lossySync} \langle d(m), \tau(m) \rangle \equiv \begin{cases} d(a_0) = d(m_0) \wedge \langle d(a)', \tau(a)' \rangle \text{lossySync} \langle d(m)', \tau(m)' \rangle & \text{if } \tau(a_0) = \tau(m_0) \\ \langle d(a)', \tau(a)' \rangle \text{lossySync} \langle d(m), \tau(m) \rangle & \text{if } \tau(a_0) < \tau(m_0) \end{cases}$$

3. For the `syncDrain` channel between nodes a and i we have:

$$\langle d(a), \tau(a) \rangle \text{syncDrain} \langle d(i), \tau(i) \rangle \equiv \tau(a) = \tau(i)$$

4. For the `merger` consists of nodes l , m , and i we have:

$$\begin{aligned} \text{merger}(\langle d(l), \tau(l) \rangle, \langle d(m), \tau(m) \rangle, \langle d(i), \tau(i) \rangle) &\equiv \tau(l_0) \neq \tau(m_0) \wedge \\ &\left\{ \begin{array}{l} d(i_0) = d(l_0) \wedge \text{merger}(\langle d(l)', \tau(l)' \rangle, \langle d(m), \tau(m) \rangle, \langle d(i)', \tau(i)' \rangle) \quad \text{if } \tau(l_0) < \tau(m_0) \\ d(i_0) = d(m_0) \wedge \text{merger}(\langle d(l), \tau(l) \rangle, \langle d(m)', \tau(m)' \rangle, \langle d(i)', \tau(i)' \rangle) \quad \text{if } \tau(m_0) < \tau(l_0) \end{array} \right. \end{aligned}$$

5. For the `sync` channel between nodes l and b we have:

$$\langle d(l), \tau(l) \rangle \text{sync} \langle d(b), \tau(b) \rangle \equiv d(l) = d(b) \wedge \tau(l) = \tau(b)$$

6. For the `sync` channel between nodes m and c we have:

$$\langle d(m), \tau(m) \rangle \text{sync} \langle d(c), \tau(c) \rangle \equiv d(m) = d(c) \wedge \tau(m) = \tau(c)$$

Now by composing the above six formulas and following the approach in section 2.7 we have:

$$\begin{aligned} \text{exclusiveRouter}(\langle d(a), \tau(a) \rangle, \langle d(b), \tau(b) \rangle, \langle d(c), \tau(c) \rangle) &\equiv \\ (d(a) = d(b) \wedge \tau(a) = \tau(b)) \otimes (d(a) = d(c) \wedge \tau(a) = \tau(c)) &\quad (4-2) \end{aligned}$$

Figure 4.2(a) shows the `Delay` connector. We used `sync` and `fifo1` channels and the `merger` and `exclusiveRouter` connectors to build this connector. The `sync` and `fifo1` channels are connected to output nodes b and c of `exclusiveRouter` connector, respectively. This way, when a data item is being output by the `exclusiveRouter` connector, it will flow through either the `sync` channel or the `fifo1` channel, but not both. To summarize the behaviour of the `delay` connector, we

can say that when a data item arrives at node a , it will be written to node d simultaneously or buffered in the fifo_1 channel first and then written to node e . Then, that data will be written to node f of the merger connector, which consists of nodes d , e , and f .

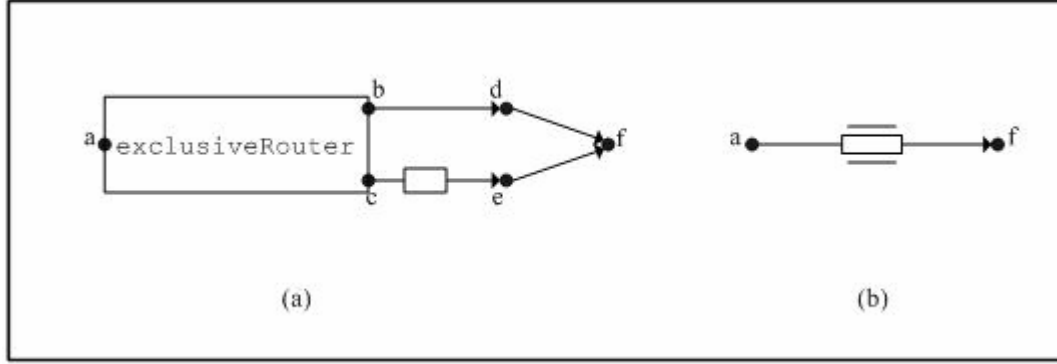


Figure 4-2 The Delay connector

From now on, we will use the notation in figure 4.2(b) whenever we want to show the Delay connector. Now we compositionally obtain its behaviour as a timed data streams relation:

1. For the sync channel between nodes b and d we have:

$$\langle d(b), \tau(b) \rangle \text{sync} \langle d(d), \tau(d) \rangle \equiv d(b) = d(d) \wedge \tau(b) = \tau(d)$$

2. For the fifo_1 channel between nodes c and e we have:

$$\langle d(c), \tau(c) \rangle \text{fifo}_1 \langle d(e), \tau(e) \rangle \equiv d(c) = d(e) \wedge \tau(c) < \tau(e) < \tau(e)'$$

3. For the merger consisting of nodes d , e , and f we have:

$$\begin{aligned} \text{merger}(\langle d(d), \tau(d) \rangle, \langle d(e), \tau(e) \rangle, \langle d(f), \tau(f) \rangle) &\equiv \tau(d_0) \neq \tau(e_0) \wedge \\ &\begin{cases} d(d_0) = d(f_0) \wedge \text{merger}(\langle d(d)', \tau(d)' \rangle, \langle d(e), \tau(e) \rangle, \langle d(f)', \tau(f)' \rangle) & \text{if } \tau(d_0) < \tau(e_0) \\ d(e_0) = d(f_0) \wedge \text{merger}(\langle d(d), \tau(d) \rangle, \langle d(e)', \tau(e)' \rangle, \langle d(f)', \tau(f)' \rangle) & \text{if } \tau(e_0) < \tau(d_0) \end{cases} \end{aligned}$$

Note that since we have used the `exclusiveRouter`, the condition $\tau(d_0) \neq \tau(e_0)$ is always satisfied.

Now by composing the formulas (1) to (3) following the approach in section 2.7 we have:

$$\langle d(a), \tau(a) \rangle_{\text{Delay}} \langle d(f), \tau(f) \rangle \equiv d(a) = d(f) \wedge \tau(a) \leq \tau(f) \quad (4-3) \blacksquare$$

In this section, in addition to introduction of a new connector, `Delay`, we explained our approach to compositionally derive the timed data stream formulas for connectors, which we are following throughout this thesis. Henceforth, we present the specification and implementation of workflow patterns.

4.2. Specification and Implementation of Basic Patterns

In the following section, we present the specification and implementation of workflow basic patterns.

4.2.1. Sequential Routing

This pattern represents a situation in a workflow where a sequence of activities is executed one after another in a single thread of execution. This means that each activity becomes enabled only when the previous activity completes. This pattern is also known as *Sequence*. As an example, in an online book ordering system, sending the bill is performed after shipping books.

4.2.1.1. Workflow Sequential Routing

Figure 4-3 represents workflow Sequential Routing pattern in which activity *B* is executed after the completion of activity *A*.

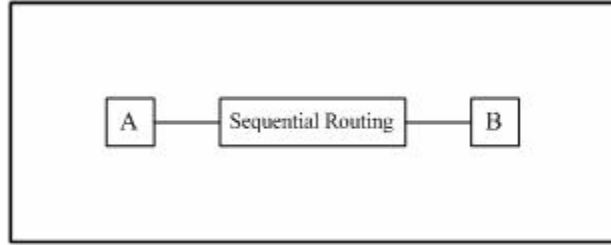


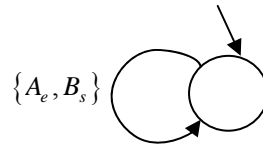
Figure 4-3 workflow Sequential Routing pattern

Formula (4-4) represents the PITL formula for this pattern.

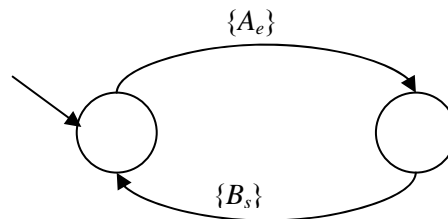
$$(A \text{ Before } B) \vee (A \text{ Meets } B) \equiv A_e < B_s \vee A_e = B_s \equiv A_e \leq B_s \quad (4-4)$$

The above formula states that activity A should always be completed either some time before or at the same time activity B becomes enabled. We, now, convert the above formula into a constraint automaton.

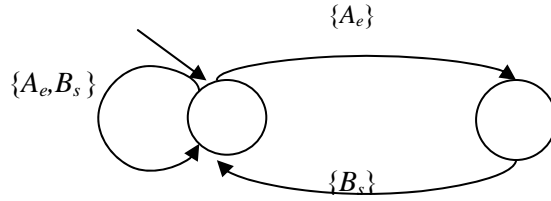
- $A_e = B_s$: since the end of activity A happens at the same time with the start of activity B there is only one transition to the same state that has both A_e, B_s as label.



- $A_e < B_s$: since the end of activity A happens before the start of activity B , one transition goes to a *waiting* state until another transition fires when activity B starts.



The resulting constraint automata for $A_e \leq B_s$ by applying the composition is:



4.2.1.2. Reo Sequential Routing

Figure 4-4 represents the implementation of Sequential Routing pattern in Reo. We implemented this pattern by simply connecting the output node of activity A to the input node of activity B using a Delay connector.

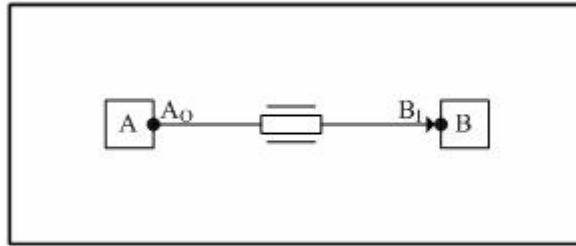
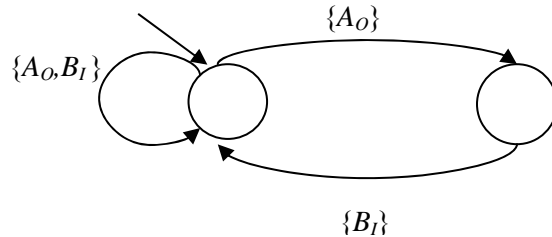


Figure 4-5 Reo Sequential Routing

Now we present the TDS behavior of the above connector. From formula (4-3) we have:

$$\begin{aligned} \langle d(A_o), \tau(A_o) \rangle \text{SequentialRouting} \langle d(B_i), \tau(B_i) \rangle &\equiv (4-5) \\ \langle d(A_o), \tau(A_o) \rangle \text{Delay} \langle d(B_i), \tau(B_i) \rangle &\equiv d(A_o) = d(B_i) \wedge \tau(A_o) \leq \tau(B_i) \end{aligned}$$

The constraint automaton for the above circuit by composing the constraint automata for the primitive channels is:



which is obviously equal to the constraint automaton of workflow Sequential Routing.

4.2.2. AND-Split

This pattern represents a situation in a workflow where a single thread of control splits into multiple parallel threads so that two or more activities can be executed in parallel or in any order. This pattern is also known as *Parallel Split*. As an example, in an online book ordering system, after the customer pays the amount of order, books will be shipped and the receipt will be sent to the customer.

4.2.2.1. Workflow AND-Split

Figure 4-5 represents workflow AND-Split pattern in which after completion of activity *A*, both activities *B* and *C* should become enabled.

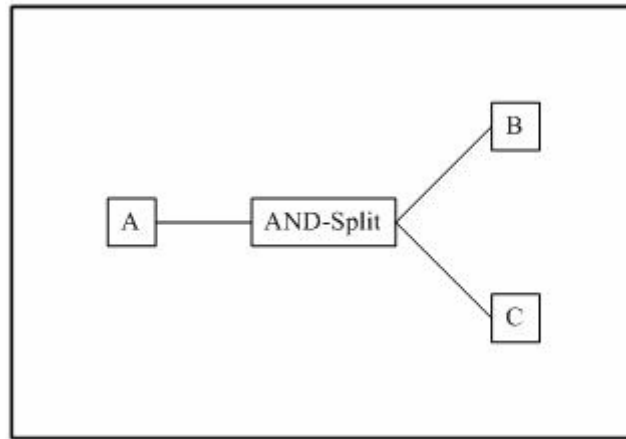


Figure 4-7 workflow AND-Split pattern

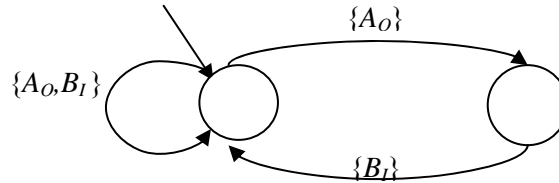
Formula (4-6) represents the PITL formula for this pattern.

$$\begin{aligned} & [(A \text{ Before } B) \vee (A \text{ Meets } B)] \wedge [(A \text{ Before } C) \vee (A \text{ Meets } C)] \equiv \\ & [A_e < B_s \vee A_e = B_s] \wedge [A_e < C_s \vee A_e = C_s] \equiv A_e \leq B_s \wedge A_e \leq C_s \end{aligned} \quad (4-6)$$

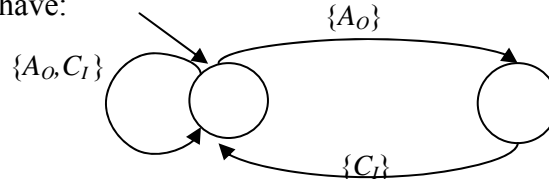
The above formula states that activity *A* should always be completed some time before or at the same time activity *B* becomes enabled and, also, some time before or at

the same time activity C becomes enabled. We, now, convert the above formula into a constraint automaton.

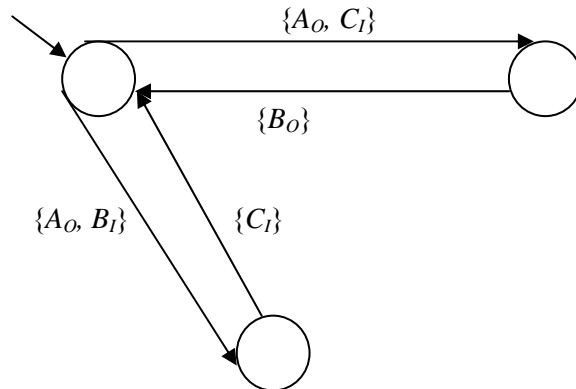
- $A_e \leq B_s$: using the same approach that we used for workflow Sequential Routing, we have:



- $A_e \leq C_s$: using the same approach that we used for workflow Sequential Routing, we have:



The resulting constraint automata for $A_e \leq B_s \wedge A_e \leq C_s$ is:



4.2.2.2.Reo AND-Split

Figure 4-6 represents the implementation of AND-Split pattern in Reo. We implemented AND-Split using Reo's replicator and Delay connectors. When activity A finishes its execution, it writes the control data on its output node which

replicates to both nodes i and j simultaneously. By using the Delay, activities B and C both become enabled either at the same time or in any order.

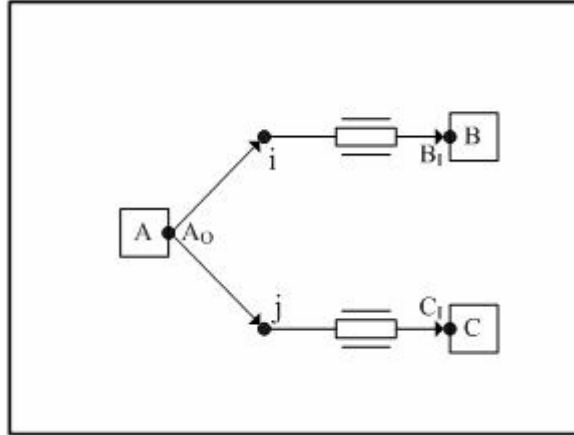


Figure 4-10 Reo AND-Split

Now we present the TDS behavior of the AND-Split connector.

1. For the replicator connector consisting of nodes A_O , i , and j we have:

$$\text{relicator}(\langle d(A_O), \tau(A_O) \rangle, \langle d(i), \tau(i) \rangle, \langle d(j), \tau(j) \rangle) \equiv \\ d(A_O) = d(i) = d(j) \wedge \tau(A_O) = \tau(i) = \tau(j)$$

2. For the Delay channel between nodes i and B_I we have:

$$\langle d(i), \tau(i) \rangle \text{Delay} \langle d(B_I), \tau(B_I) \rangle \equiv d(i) = d(B_I) \wedge \tau(i) \leq \tau(B_I)$$

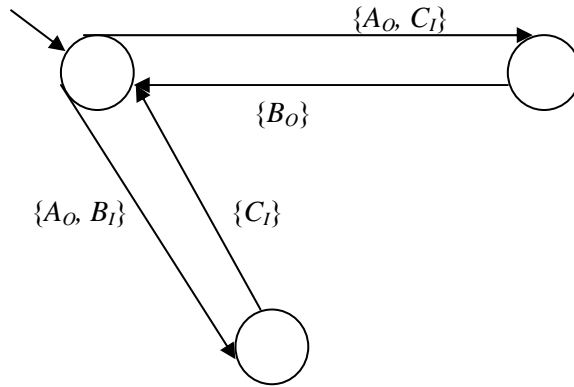
3. For the Delay channel between nodes j and C_I we have:

$$\langle d(j), \tau(j) \rangle \text{Delay} \langle d(C_I), \tau(C_I) \rangle \equiv d(j) = d(C_I) \wedge \tau(j) \leq \tau(C_I)$$

By composing three formulas above, we obtain the following TDS formula for Reo AND-Split connector:

$$\text{AND-Split}(\langle d(A_o), \tau(A_o) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv d(A_o) = d(B_I) = d(C_I) \wedge \tau(A_o) \leq \tau(B_I) \wedge \tau(A_o) \leq \tau(C_I) \quad (4-7)$$

The constraint automaton for the above circuit is:



This is obviously equal to the constraint automata of workflow AND-Split.

4.2.3.AND-Join

This pattern represents a situation in a workflow where multiple parallel activities converge into one thread and are synchronized. This pattern is also known as *Synchronization*. Patterns *AND-Split* and *AND-Join* together are known as *Parallel Routing*. As an example, in an online ticket ordering system after the tickets are sent and the payment is received then the information of the sold tickets will be archived.

4.2.3.1. Workflow AND-Join

Figure 4-7 represents an instance of workflow AND-Join pattern in which after completion of both activities *A* and *B*, activity *C* should become enabled.

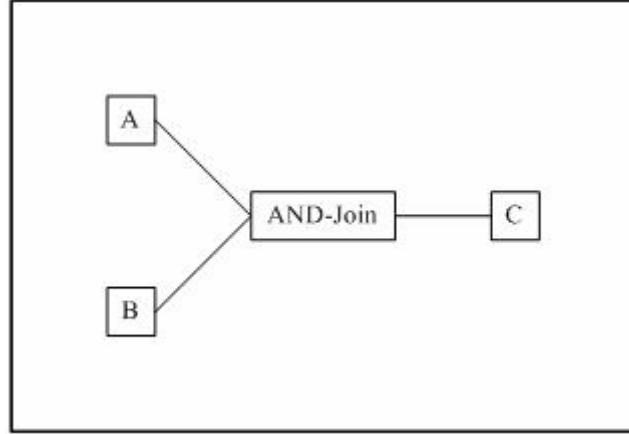


Figure 4-13 workflow AND-Join pattern

Formula (4-8) represents the PCTL formula for this pattern.

$$\begin{aligned}
 & [(A \text{ Before } C) \vee (A \text{ Meets } C)] \wedge [(B \text{ Before } C) \vee (B \text{ Meets } C)] \equiv \\
 & [A_e < C_s \vee A_e = C_s] \wedge [B_e < C_s \vee B_e = C_s] \equiv A_e \leq C_s \wedge B_e \leq C_s \quad (4-8)
 \end{aligned}$$

The above formula states that activities *A* and *B* should always be completed either some time before or at the same time activity *C* becomes enabled.

4.2.3.2. Reo AND-Join

Figure 4-8 represents the implementation of AND-Join pattern in Reo. We implemented AND-Join using the `syncDrain` channel and `Delay` connector. Depending on which *A* or *B* finishes executing first, its respective `write` operation on its output node will remain pending until the other activity finishes its execution and becomes ready to perform `write` on its output node. Then all `writes` will be performed simultaneously and succeed. Then, according to `Delay` connector, activity *C* will become enabled whether immediately or later.

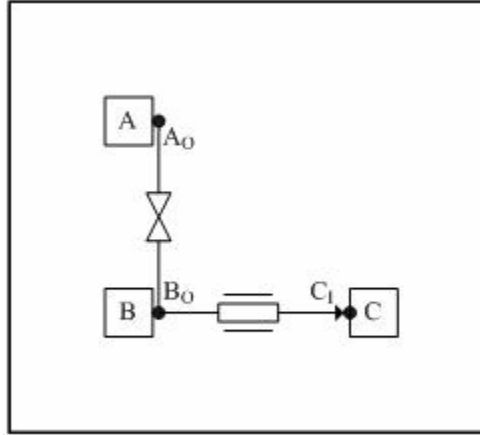


Figure 4-14 Reo AND-Join connector

Now we present the TDS behavior of the AND-Join connector.

1. For the `syncDrain` channel between nodes A_O and B_O we have:

$$\langle d(A_O), \tau(A_O) \rangle \text{syncDrain} \langle d(B_O), \tau(B_O) \rangle \equiv \tau(A_O) = \tau(B_O)$$

2. For the `Delay` channel between nodes B_O and C_I we have:

$$\begin{aligned} \langle d(B_O), \tau(B_O) \rangle \text{Delay} \langle d(C_I), \tau(C_I) \rangle \equiv \\ d(B_O) = d(C_I) \wedge \tau(B_O) \leq \tau(C_I) \end{aligned}$$

By composing two above formulas, we obtain the TDS formula for Reo AND-Join connector:

$$\begin{aligned} \text{AND-Join}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \\ d(B_O) = d(C_I) \wedge \tau(A_O) \leq \tau(C_I) \wedge \tau(B_O) \leq \tau(C_I) \end{aligned} \quad (4-9)$$

4.2.4. XOR-Split

This pattern represents a situation in a workflow where among several alternative activities, one is chosen based on some conditions or control data. This pattern is also known as *Exclusive Choice*. As an example, in an insurance claim processing system, the

evaluation of the claim is followed by either paying the damage or contacting the customer, but not both.

4.2.4.1. Workflow XOR-Split

Figure 4-9 represents workflow XOR-Split pattern in which after completion of activity *A*, based on whether condition *cond1* or *cond2* is true, either activity *B* or activity *C* becomes enabled, respectively. Note that we assume *cond1* and *cond2* are zero-length intervals at which conditions *cond1* and *cond2* become true, respectively.

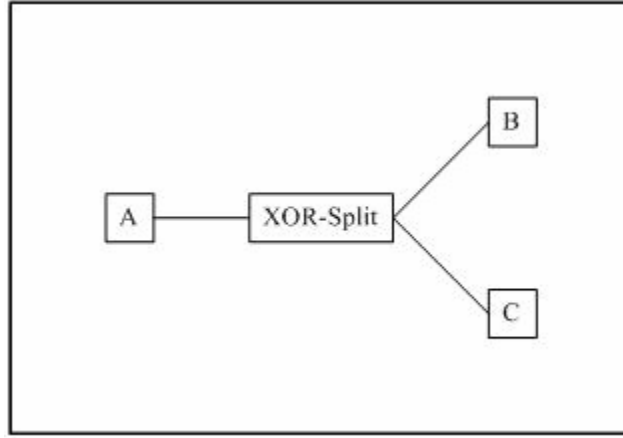


Figure 4-15 workflow XOR-Split Pattern

Formula (4-10) represents the PITL formula for this pattern. Note that *cond1* and *cond2* are exclusive conditions.

$$\begin{aligned}
 & \left[(cond1 \mathbf{F}inishes A) \rightarrow (A \mathbf{B}efore B) \vee (A \mathbf{M}eets B) \right] \wedge & (4-10) \\
 & \left[(cond2 \mathbf{F}inishes A) \rightarrow (A \mathbf{B}efore C) \vee (A \mathbf{M}eets C) \right] \equiv \\
 & \left[A_e = cond1 \rightarrow (A_e < B_s \vee A_e = B_s) \right] \wedge \left[A_e = cond2 \rightarrow (A_e < C_s \vee A_e = C_s) \right] \equiv \\
 & (A_e = cond1 \rightarrow A_e \leq B_s) \wedge (A_e = cond2 \rightarrow A_e \leq C_s)
 \end{aligned}$$

The above formula states that if at the time activity *A* is completed, *cond1* is true then the completion of activity *A* happens some time before or at the same time activity *B* becomes enabled; but if at the time activity *A* is completed, *cond2* is true then the

completion of activity *A* happens some time before or at the same time activity *C* becomes enabled.

4.2.4.2. Reo XOR-Split

Figure 4-10 represents the implementation of XOR-Split pattern in Reo. We implemented the XOR-Split pattern using the Delay connector and two `filter` channels. When activity *A* finishes its execution, it replicates the token on both `filter` channels connected to its output node. If the data written by *A* matches with the patterns of any of the `filter` channels, the corresponding activity attached to that channel will become enabled and the other channel will lose the data. It is the designer's task to specify *complete* and *exclusive* patterns.

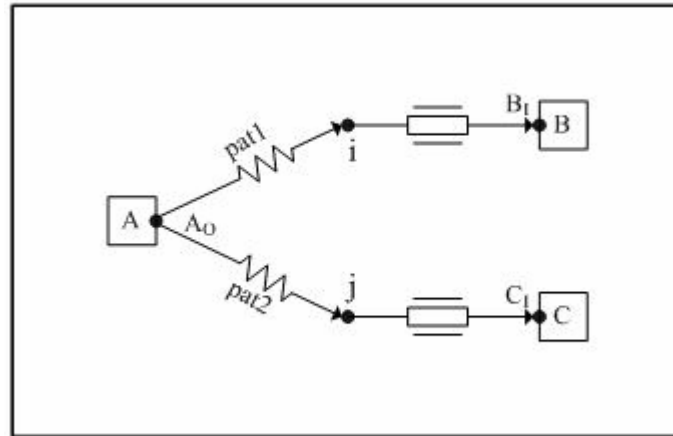


Figure 4-16 Reo XOR-Split connector

Now we present the TDS behavior of the XOR-Split connector.

1. For the `filter` channel between nodes A_o and i we have:

$$\langle d(A_o), \tau(A_o) \rangle \text{filter}(\text{pat1}) \langle d(i), \tau(i) \rangle \equiv \begin{cases} d(A_o) = d(i) \wedge \tau(A_o) = \tau(i) \wedge \\ \langle d(A_o)', \tau(A_o)' \rangle \text{filter}(\text{pat1}) \langle d(i)', \tau(i)' \rangle & \text{if } \text{pat1} \\ \langle d(A_o)', \tau(A_o)' \rangle \text{filter}(\text{pat1}) \langle d(i), \tau(i) \rangle & \text{if } \neg \text{pat1} \end{cases}$$

2. For the `filter` channel between nodes A_O and j we have:

$$\langle d(A_O), \tau(A_O) \rangle \text{filter}(\text{pat2}) \langle d(j), \tau(j) \rangle \equiv \begin{cases} d(A_{O0}) = d(j_0) \wedge \tau(A_{O0}) = \tau(j_0) \wedge \\ \langle d(A_O)', \tau(A_O)' \rangle \text{filter}(\text{pat2}) \langle d(j)', \tau(j)' \rangle & \text{if } \text{pat2} \\ \langle d(A_O)', \tau(A_O)' \rangle \text{filter}(\text{pat2}) \langle d(j), \tau(j) \rangle & \text{if } \neg \text{pat2} \end{cases}$$

3. For the `Delay` channel between nodes i and B_I we have:

$$\langle d(i), \tau(i) \rangle \text{Delay} \langle d(B_I), \tau(B_I) \rangle \equiv d(i) = d(B_I) \wedge \tau(i) \leq \tau(B_I)$$

4. For the `Delay` channel between nodes j and C_I we have:

$$\langle d(j), \tau(j) \rangle \text{Delay} \langle d(C_I), \tau(C_I) \rangle \equiv d(j) = d(C_I) \wedge \tau(j) \leq \tau(C_I)$$

Note that we assume that `pat1` and `pat2` are exclusive conditions. By composing four formulas above, we obtain the TDS formula for Reo XOR-Split connector:

$$\text{XOR-Split}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \begin{cases} d(A_O) = d(B_I) \wedge \tau(A_O) \leq \tau(B_I) & \text{if } \text{pat1} \\ d(A_O) = d(C_I) \wedge \tau(A_O) \leq \tau(C_I) & \text{if } \text{pat2} \end{cases} \quad (4-11)$$

4.2.5. XOR-Join

This pattern represents a situation in a workflow when one out of several alternative branches completes. It is assumed that branches are never executed in parallel. This pattern is also known as *Simple Merge*. As an example, in an insurance claim processing system, archiving the claim is done after either paying the damage or contacting the customer.

4.2.5.1. Workflow XOR-Join

Figure 4-11 represents workflow XOR-Join pattern in which after completion of either activities *A* or *B*, activity *C* should become enabled.

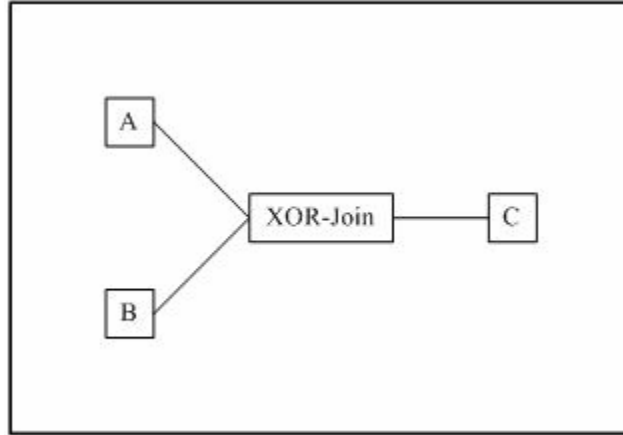


Figure 4-17 workflow XOR-Join pattern

Formula (4-12) represents the PITL formula for this pattern. There is an assumption that *A* and *B* never being executed in parallel.

$$\begin{aligned} [(A \text{ Before } C) \vee (A \text{ Meets } C)] \vee [(B \text{ Before } C) \vee (B \text{ Meets } C)] &\equiv (4-12) \\ [A_e < C_s \vee A_e = C_s] \vee [B_e < C_s \vee B_e = C_s] &\equiv A_e \leq C_s \vee B_e \leq C_s \end{aligned}$$

The above formula states that completion of any of activities *A* or *B* happens some time before or at the same time activity *C* becomes enabled.

4.2.5.2. Reo XOR-Join

Figure 4-12 represents the implementation of AND-Join pattern in Reo. We implemented the XOR-Join pattern using Reo's `merger` and `Delay` connectors. In this circuit, node C_I will take the token provided by either *A* or *B*, or if they both have their token ready, it will choose one nondeterministically. In this pattern, we assume that activity *A* and *B* are never executed in parallel; the designer is responsible for this assumption.

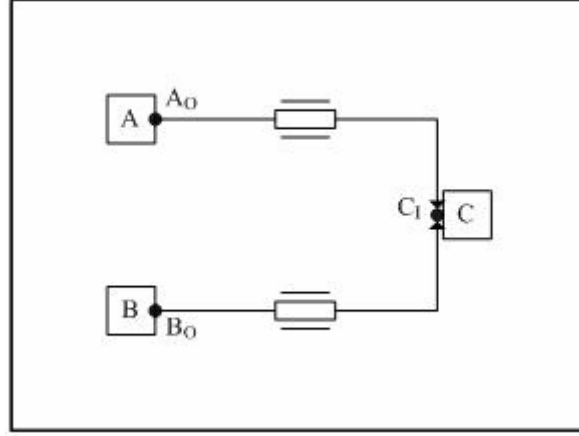


Figure 4-18 Reo XOR-Join connector

Now we present the TDS behavior of the XOR-Join connector.

1. For the Delay channel between nodes A_O and C_I we have:

$$\langle d(A_O), \tau(A_O) \rangle \text{Delay} \langle d(C_I), \tau(C_I) \rangle \equiv d(A_O) = d(C_I) \wedge \tau(A_O) \leq \tau(C_I)$$

2. For the Delay channel between nodes B_O and C_I we have:

$$\langle d(B_O), \tau(B_O) \rangle \text{Delay} \langle d(C_I), \tau(C_I) \rangle \equiv d(B_O) = d(C_I) \wedge \tau(B_O) \leq \tau(C_I)$$

3. For the merger consisting of nodes A_O , B_O , and C_I we have:

$$\text{merger}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \tau(A_{O0}) \neq \tau(B_{O0}) \wedge \left\{ \begin{array}{l} d(A_{O0}) = d(C_{I0}) \wedge \tau(A_{O0}) \leq \tau(C_{I0}) \wedge \\ \text{merger}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(C_I)', \tau(C_I)' \rangle) \quad \text{if } \tau(A_{O0}) < \tau(B_{O0}) \\ d(B_{O0}) = d(C_{I0}) \wedge \tau(B_{O0}) \leq \tau(C_{I0}) \wedge \\ \text{merger}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_O)', \tau(B_O)' \rangle, \langle d(C_I)', \tau(C_I)' \rangle) \quad \text{if } \tau(B_{O0}) < \tau(A_{O0}) \end{array} \right.$$

By composing two formulas above and definition of merger connector, we obtain the TDS formula for Reo XOR-Join connector:

$$\begin{aligned}
& \text{XOR-Join}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv & (4-13) \\
& \tau(A_{O0}) \neq \tau(B_{O0}) \wedge \\
& \left\{ \begin{array}{l} d(A_{O0}) = d(C_{I0}) \wedge \tau(A_{O0}) \leq \tau(C_{I0}) \wedge \\ \text{XOR-Join}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(C_I)', \tau(C_I)' \rangle) \quad \text{if } \tau(A_{O0}) < \tau(B_{O0}) \\ d(B_{O0}) = d(C_{I0}) \wedge \tau(B_{O0}) \leq \tau(C_{I0}) \wedge \\ \text{XOR-Join}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_O)', \tau(B_O)' \rangle, \langle d(C_I)', \tau(C_I)' \rangle) \quad \text{if } \tau(B_{O0}) < \tau(A_{O0}) \end{array} \right.
\end{aligned}$$

4.3. Specification and Implementation of Advanced Synchronization Patterns

In the following section, we present the specification and implementation of workflow advanced synchronization patterns.

4.3.1. OR-Split

This pattern represents a situation in a workflow when, based on some conditions or control data, various numbers of branches among several alternatives are chosen. This pattern is also known as *Multi Choice*. As an example, in an insurance claim processing system, after evaluating the damage, then the fire department or insurance company will be contacted. At least one of these activities is executed. However, it is also possible that both need to be executed.

4.3.1.1. Workflow OR-Split

Figure 4-13 represents workflow XOR-Split pattern in which after completion of activity *A*, based on whether condition *cond1* or *cond2* or both are true, either activity *B* or *C* or both become enabled, respectively. Note that we assume *cond1* and *cond2* are zero-length intervals at which conditions *cond1* and *cond2* become true, respectively.

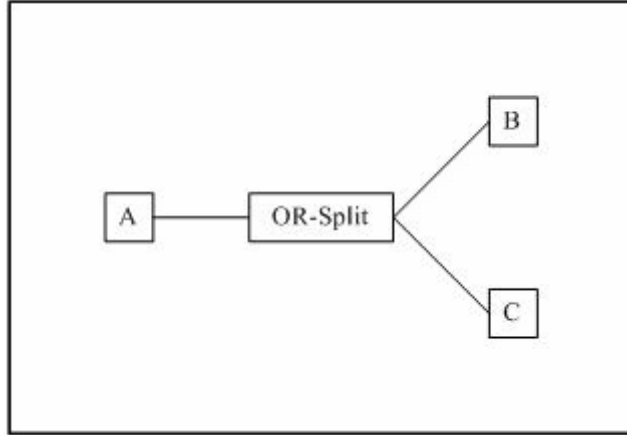


Figure 4-19 workflow OR-Split pattern

Formula (4-14) represents the PITL formula for this pattern. Note that *cond1* and *cond2* are inclusive conditions.

$$\left[(cond1 \text{ Finishes } A) \rightarrow (A \text{ Before } B) \vee (A \text{ Meets } B) \right] \wedge \quad (4-13)$$

$$\left[(cond2 \text{ Finishes } A) \rightarrow (A \text{ Before } C) \vee (A \text{ Meets } C) \right] \wedge$$

$$\left[((cond1 \wedge cond2) \text{ Finishes } A) \rightarrow$$

$$\left((A \text{ Before } C) \vee (A \text{ Meets } C) \right) \wedge \left((A \text{ Before } B) \vee (A \text{ Meets } B) \right) \right] \equiv$$

$$\left[A_e = cond1 \rightarrow (A_e < B_s \vee A_e = B_s) \right] \wedge \left[A_e = cond2 \rightarrow (A_e < C_s \vee A_e = C_s) \right] \wedge$$

$$\left[A_e = cond1 \wedge A_e = cond2 \rightarrow \left((A_e < B_s \vee A_e = B_s) \wedge (A_e < C_s \vee A_e = C_s) \right) \right] \equiv$$

$$(A_e = cond1 \rightarrow A_e \leq B_s) \wedge (A_e = cond2 \rightarrow A_e \leq C_s) \wedge (A_e = cond1 \wedge A_e = cond2 \rightarrow A_e \leq B_s \wedge A_e \leq C_s)$$

The above formula states that if at the time activity *A* is completed, *cond1* is true then the completion of activity *A* happens some time before or at the same time activity *B* becomes enabled; if at the time activity *A* is completed, *cond2* is true then the completion of activity *A* happens some time before or at the same time activity *C* becomes enabled. Finally, if at the time activity *A* is completed, both *cond1* and *cond2* are true, then the completion of activity *A* happens some time before or at the same time both activities *B* and *C* become enabled.

4.3.1.2. Reo OR-Split

Figure 4-14 represents the implementation of OR-Split pattern in Reo. We used the same structure to implement the OR-Split pattern as the one we used for the XOR-Split pattern. The only difference here is that patterns for `filter` channels should be *complete* and *inclusive*.

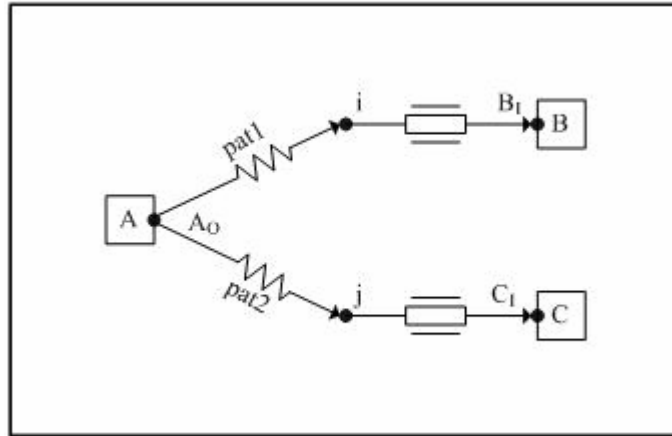


Figure 4-20 Reo OR-Split connector

Now we present the TDS behavior of the above connector.

1. For the `filter` channel between nodes A_O and i we have:

$$\langle d(A_O), \tau(A_O) \rangle \text{filter}(\text{pat2}) \langle d(i), \tau(i) \rangle \equiv \begin{cases} d(A_{O0}) = d(i_0) \wedge \tau(A_{O0}) = \tau(i_0) \wedge \\ \langle d(A_O)', \tau(A_O)' \rangle \text{filter}(\text{pat2}) \langle d(i_0)', \tau(i_0)' \rangle & \text{if } \text{pat2} \\ \langle d(A_O)', \tau(A_O)' \rangle \text{filter}(\text{pat2}) \langle d(i_0), \tau(i_0) \rangle & \text{if } \neg \text{pat2} \end{cases}$$

2. For the `filter` channel between nodes A_O and j we have:

$$\langle d(A_o), \tau(A_o) \rangle \text{filter}(\text{pat2}) \langle d(j), \tau(j) \rangle \equiv \begin{cases} d(A_{o0}) = d(j_0) \wedge \tau(A_{o0}) = \tau(j_0) \wedge \\ \langle d(A_o)', \tau(A_o)' \rangle \text{filter}(\text{pat2}) \langle d(j)', \tau(j)' \rangle & \text{if } \text{pat2} \\ \langle d(A_o)', \tau(A_o)' \rangle \text{filter}(\text{pat2}) \langle d(j), \tau(j) \rangle & \text{if } \neg \text{pat2} \end{cases}$$

3. For the Delay channel between nodes i and B_I we have:

$$\langle d(i), \tau(i) \rangle \text{Delay} \langle d(B_I), \tau(B_I) \rangle \equiv d(i) = d(B_I) \wedge \tau(i) \leq \tau(B_I)$$

4. For the Delay channel between nodes j and C_I we have:

$$\langle d(j), \tau(j) \rangle \text{Delay} \langle d(C_I), \tau(C_I) \rangle \equiv d(j) = d(C_I) \wedge \tau(j) \leq \tau(C_I)$$

Note that we assume that pat1 and pat2 are inclusive conditions. By composing the four formulas above, we obtain the TDS formula for Reo OR-Split connector:

$$\text{OR-Split}(\langle d(A_o), \tau(A_o) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \begin{cases} d(A_o) = d(B_I) \wedge \tau(A_o) \leq \tau(B_I) & \text{if } \text{pat1} \\ d(A_o) = d(C_I) \wedge \tau(A_o) \leq \tau(C_I) & \text{if } \text{pat2} \\ d(A_o) = d(B_I) = d(C_I) \wedge \tau(A_o) \leq \tau(B_I) \wedge \tau(A_o) \leq \tau(C_I) & \text{if } \text{pat1} \wedge \text{pat2} \end{cases} \quad (4-14)$$

4.3.2. Synchronizing Merge

This pattern represents a situation in a workflow when several branches converge into one single thread. If more than one branch is active, it synchronizes the active branches and merges other branches. Deciding when to merge and when to synchronize is a runtime decision. It is assumed that a branch that has already been activated cannot be activated again while the merge is still waiting for other branches to complete. As an example, in an insurance claim processing system, one or both contacting the fire department and insurance company have been completed, a report will be submitted.

4.3.2.1. Workflow Synchronizing Merge

Figure 4-15 represents workflow Synchronizing Merge pattern in which if after completion of activity *A*, only one of activities *B* or *C* becomes enabled, then, after its completion activity *D* becomes enabled. If after completion of activity *A*, both activities *B* and *C* become enabled, then they should be synchronized so that activity *D* can be enabled next. In figure 4-15, we used the OR-Join pattern to show that either activity *B*, or *C* or both can become enabled.

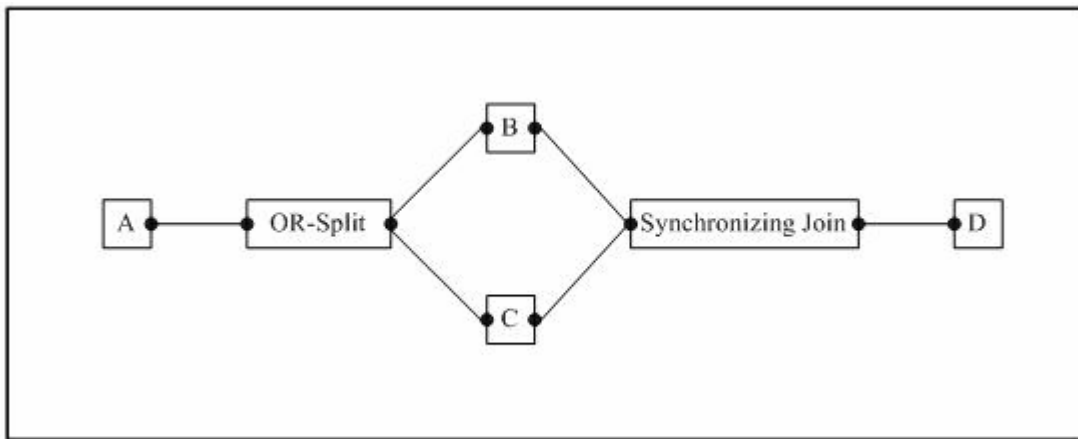


Figure 4-21 workflow synchronizing merge pattern

Formula (4-15) represents the PITL formula for this pattern.

$$\begin{aligned}
 & \left[\left((A \text{ Before } B) \vee (A \text{ Meets } B) \right) \wedge \neg C \rightarrow (B \text{ Before } D) \vee (B \text{ Meets } D) \right] \wedge & (4-15) \\
 & \left[\left((A \text{ Before } C) \vee (A \text{ Meets } C) \right) \wedge \neg B \rightarrow (C \text{ Before } D) \vee (C \text{ Meets } D) \right] \wedge \\
 & \left[\left((A \text{ Before } B) \vee (A \text{ Meets } B) \right) \wedge \left((A \text{ Before } C) \vee (A \text{ Meets } C) \right) \rightarrow \right. \\
 & \left. \left((B \text{ Before } D) \vee (B \text{ Meets } D) \right) \wedge \left((C \text{ Before } D) \vee (C \text{ Meets } D) \right) \right] \equiv \\
 & \left[(A_e < B_s \vee A_e = B_s) \wedge \neg C \rightarrow (B_e < D_s \vee B_e = D_s) \right] \wedge \\
 & \left[(A_e < C_s \vee A_e = C_s) \wedge \neg B \rightarrow (C_e < D_s \vee C_e = D_s) \right] \wedge \\
 & \left[\left((A_e < B_s \vee A_e = B_s) \wedge (A_e < C_s \vee A_e = C_s) \right) \rightarrow \left((B_e < D_s \vee B_e = D_s) \wedge (C_e < D_s \vee C_e = D_s) \right) \right] \equiv \\
 & (A_e \leq B_s \wedge \neg C \rightarrow B_e \leq D_s) \wedge (A_e \leq C_s \wedge \neg B \rightarrow C_e \leq D_s) \wedge \left((A_e \leq B_s \wedge A_e \leq C_s) \rightarrow (B_e \leq D_s \wedge C_e \leq D_s) \right)
 \end{aligned}$$

The above formula represents three situations for Synchronizing Merge patterns:

- The case when activity *B* becomes enabled but not activity *C*: if completion of activity *A* happens some time before or at the same time with activity *B*, then completion of *B* happens some time before or at the same time with activity *D*.
- The case when activity *C* becomes enabled but not activity *B*: if completion of activity *A* happens some time before or at the same time with activity *C*, then completion of *C* happens some time before or at the same time with activity *D*.
- The case when both activities *B* and *C* become enabled: if completion of activity *A* happens some time before or at the same time with both activities *B* and *C*, then completion of *B* and *C* should happen some time before or at the same time with activity *D*.

4.3.2.2. Reo Synchronizing Merge

The problem with synchronizing merge is that we need to decide when to synchronize and when to merge. This decision should be made at run time. We implement the synchronizing merge pattern by using `AND-Split`, `AND-Join`, and `Delay` connectors. Figure 4-16 represents the implementation of Synchronizing Merge pattern in Reo.

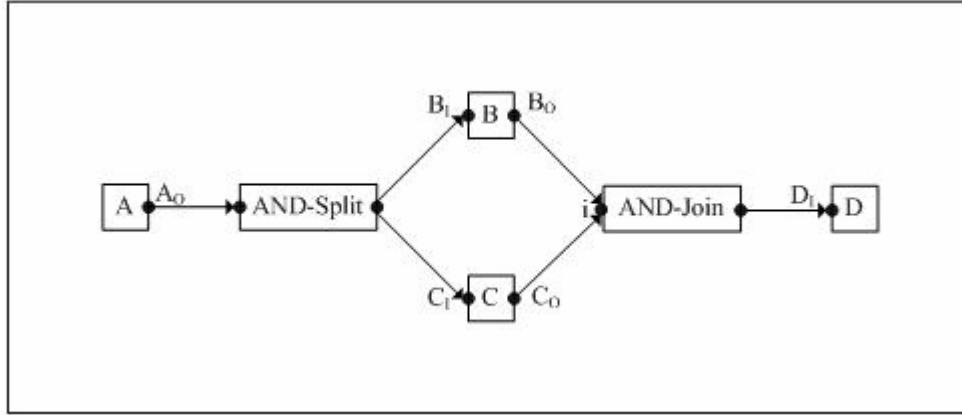


Figure 4-22 Reo Synchronizing Merge connector

When activity A finishes its execution, it replicates the token through the AND-Split connector. If only one of the activities should become enabled, e.g. B , the other activity, e.g. C will consider that token as a *dummy* data and will write it on its output node without being executed. After activity B finishes executing, both B and C will be synchronized using the AND-Join connector. Then, activity D will become enabled either simultaneously or later.

Now we present the TDS behavior of the Synchronizing Merge connector:

1. For the AND-Split connector consists of nodes A_o , B_i , and C_i we have:

$$\text{AND-Split}(\langle d(A_o), \tau(A_o) \rangle, \langle d(B_i), \tau(B_i) \rangle, \langle d(C_i), \tau(C_i) \rangle) \equiv \\ d(A_o) = d(B_i) = d(C_i) \wedge \tau(A_o) \leq \tau(B_i) \wedge \tau(A_o) \leq \tau(C_i)$$

2. For the AND-Join connector we have:

$$\text{AND-Join}(\langle d(B_o), \tau(B_o) \rangle, \langle d(C_o), \tau(C_o) \rangle, \langle d(D_i), \tau(D_i) \rangle) \equiv \\ d(B_o) = d(C_o) = d(D_i) \wedge \tau(B_o) \leq \tau(D_i) \wedge \tau(C_o) \leq \tau(D_i)$$

By composing two formulas above, we obtain the TDS formula for Reo Synchronizing Merge connector:

$$\begin{aligned} \text{syncMerge}(\langle d(A_o), \tau(A_o) \rangle, \langle d(B_i), \tau(B_i) \rangle, \langle d(C_i), \tau(C_i) \rangle, & \quad (4-16) \\ \langle d(B_o), \tau(B_o) \rangle, \langle d(C_o), \tau(C_o) \rangle, \langle d(D_i), \tau(D_i) \rangle) \equiv \\ [d(A_o) = d(B_i) = d(C_i) \wedge \tau(A_o) \leq \tau(B_i) \wedge \tau(A_o) \leq \tau(C_i)] \wedge \\ [d(B_o) = d(C_o) = d(D_i) \wedge \tau(B_o) \leq \tau(D_i) \wedge \tau(C_o) \leq \tau(D_i)] \end{aligned}$$

4.3.3. Multi Merge

This pattern represents a situation in a workflow when several branches converge into one single thread. The activity following this merge construct, e.g. *C*, will be executed once for every completion of every active branch so allow us to present activity *C* only one time and avoid redundant appearance of activity *C*. As an example, two activities auditing and processing an application are performing in parallel and both should be followed by closing case.

4.3.3.1. Workflow Multi Merge

Figure 4-17 represents workflow Multi Merge pattern in which after completion of activities *A* and *B*, activity *C* becomes enabled once.

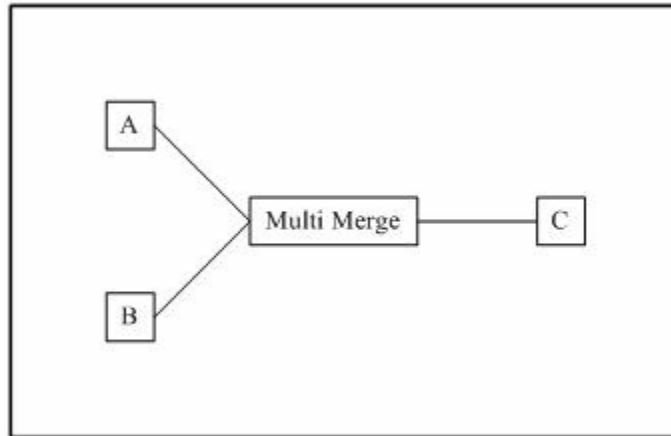


Figure 4-23 workflow Multi Merge pattern

Formula (4-17) represents the PITL formula for this pattern.

$$\begin{aligned}
& \left[(A \text{ Before } C) \vee (A \text{ Meets } C) \right] \vee \left[(B \text{ Before } C) \vee (B \text{ Meets } C) \right] \vee & (4-17) \\
& \left[\left((A \text{ Before } C) \vee (A \text{ Meets } C) \right) \wedge \left((B \text{ Before } C) \vee (B \text{ Meets } C) \right) \right] \equiv \\
& \left[A_e < C_s \vee A_e = C_s \right] \vee \left[B_e < C_s \vee B_e = C_s \right] \vee \left[(A_e < C_s \vee A_e = C_s) \wedge (B_e < C_s \vee B_e = C_s) \right] \equiv \\
& (A_e \leq C_s) \vee (B_e \leq C_s) \vee (A_e \leq C_s \wedge B_e \leq C_s)
\end{aligned}$$

The above formula states that completion of either activity A or B or both is followed either some time before or at the same time by enabling of activity C . From this formula it is obvious that after completion of each of A or B , activity C will become enabled once.

4.3.3.2. Reo Multi Merge

Figure 4-18 presents the implementation of Multi Merge pattern in Reo. We implemented this pattern using Reo's merger and Delay connectors and in the same way we implemented XOR-Join pattern. The only difference here is that activities A and B can be executed in parallel. So we have the same TDS formula.

$$\begin{aligned}
& \text{MultiMerge}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv & (4-18) \\
& \tau(A_{O0}) \neq \tau(B_{O0}) \wedge \\
& \left\{ \begin{array}{l} d(A_{O0}) = d(C_{I0}) \wedge \tau(A_{O0}) \leq \tau(C_{I0}) \wedge \\ \text{MultiMerge}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(C_I)', \tau(C_I)' \rangle) \quad \text{if } \tau(A_{O0}) < \tau(B_{O0}) \\ d(B_{O0}) = d(C_{I0}) \wedge \tau(B_{O0}) \leq \tau(C_{I0}) \wedge \\ \text{MultiMerge}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_O)', \tau(B_O)' \rangle, \langle d(C_I)', \tau(C_I)' \rangle) \quad \text{if } \tau(B_{O0}) < \tau(A_{O0}) \end{array} \right.
\end{aligned}$$

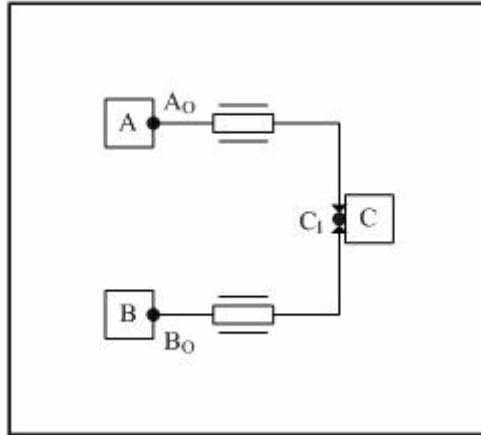


Figure 4-24 Reo Multi Merge connector

4.3.4. Discriminator

This pattern represents a situation in a workflow when several branches converge into a single thread. The activity following this merge construct will be executed once for the first activity that completes; then it waits for other alternative branches to complete and ignores them.

The Discriminator pattern can easily be generalized for the situation when an activity should be triggered only after n out of m incoming branches have been completed. Similar to the basic discriminator all remaining branches should be ignored. As an example, to improve query response time, a complex search is sent to two different databases over the Internet. The first one that comes up with the result should proceed the flow. The second result is ignored.

4.3.4.1. Workflow Discriminator

Figure 4-19 represents workflow Discriminator pattern in which if only one of activities A or B is executing or both of them are executing, then after completion of one of them, whichever finishes sooner, activity C becomes enabled only one time; the other one, in case of execution, will be completed and ignored.

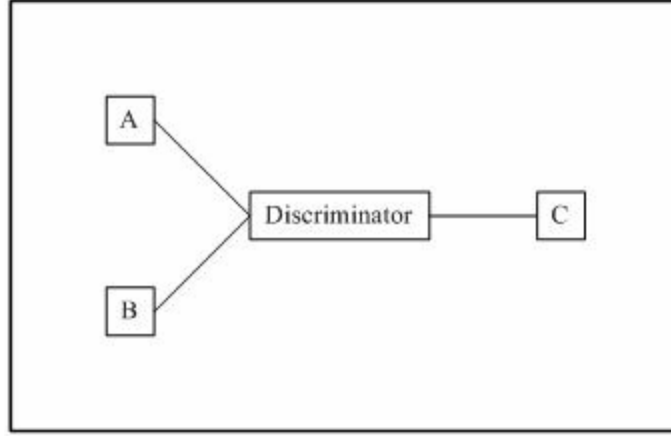


Figure 4-25 workflow discriminator pattern

Formula (4-19) represents the PITL formula for this pattern.

$$\begin{aligned}
& \left[(A \text{ Overlaps } B) \vee (A \text{ Starts } B) \vee (A \text{ During } B) \rightarrow (A \text{ Before } C) \vee (A \text{ Meets } C) \right] \wedge \\
& \left[(B \text{ Overlaps } A) \vee (B \text{ Starts } A) \vee (B \text{ During } A) \rightarrow (B \text{ Before } C) \vee (B \text{ Meets } C) \right] \wedge \\
& \left[(A \text{ Finishes } B) \vee (B \text{ Finishes } A) \vee (A \text{ Equals } B) \rightarrow (A \text{ Before } C) \vee (A \text{ Meets } C) \right] \equiv \\
& \left[(A_s < B_s \wedge B_s < A_e \wedge A_e < B_e) \vee (A_s = B_s \wedge A_e < B_e) \vee (A_s > B_s \wedge A_e < B_e) \rightarrow (A_e \leq C_s) \right] \wedge \\
& \left[(B_s < A_s \wedge A_s < B_e \wedge B_e < A_e) \vee (A_s = B_s \wedge A_e > B_e) \vee (B_s > A_s \wedge B_e < A_e) \rightarrow (B_e \leq C_s) \right] \wedge \\
& \left[(B_s \leq A_s) \vee (B_s \geq A_s) \vee (B_s = A_s \wedge B_e = A_e) \rightarrow (A_e \leq C_s) \right] \Rightarrow \\
& (A_e < B_e \rightarrow A_e \leq C_s) \wedge (B_e < A_e \rightarrow B_e \leq C_s) \wedge (A_e = B_e \rightarrow A_e \leq C_s) \quad (4-19)
\end{aligned}$$

In the above formula we show the behaviour of Discriminator pattern considering the ordering of activities A and B . That formula represents three cases in which each case consists of three parts itself regarding completion time of each of two activities;

Case 1 in which activity A finishes before activity B ; activity C becomes enabled for activity A once:

1. **A Overlaps B** : it means that A started earlier than B but finishes sooner.

2. **A Starts B**; it means that *A* started at the same time with *B* but finishes sooner.
3. **A During B**; it means that *A* started later than *B* but finishes sooner.

Case 2 in which activity *B* finishes before activity *A*; activity *C* becomes enabled for activity *B* once:

1. **B Overlaps A**: it means that *B* started earlier than *A*, but finishes sooner.
2. **B Starts A**; it means that *B* started at the same time with *A*, but finishes sooner.
3. **B During A**; it means that *B* started later than *A*, but finishes sooner.

Case 3 in which activities *A* and *B* finish at the same time; then, activity *C* becomes enabled only once:

1. **A Finishes B**: it means that *A* started earlier than *B*, but both finish at the same time.
2. **B Finishes A**; it means that *B* started earlier than *A*, but both finish at the same time.
3. **A Equal B**; it means that both *A* and *B* start and finish at the same time.

4.3.4.2. Reo Discriminator

Figure 4-20 represents the implementation of Discriminator pattern in Reo¹. This circuit works as follow. If activity *A* finishes before activity *B* it will writes the token to

¹ The implementation of this pattern has been taken from <http://homepages.cwi.nl/~proenca/webreo/>.

the A_{oi} and m_l buffers. These buffers can only be emptied if both activities B and C finish their execution.

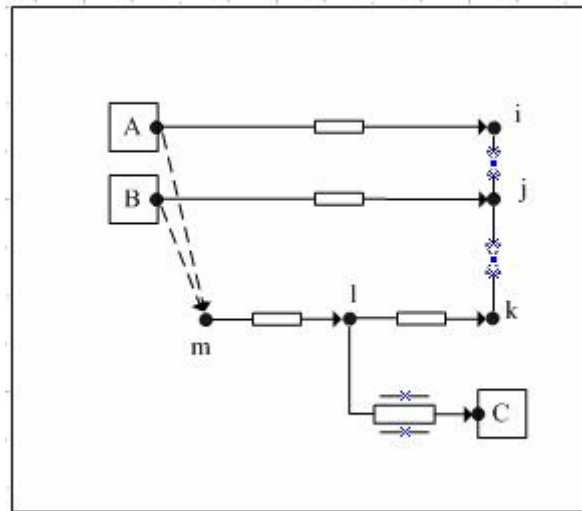


Figure 4-26 Reo Discriminator Connector

Figure 4-27 shows the “n out of m join” construct in Reo.

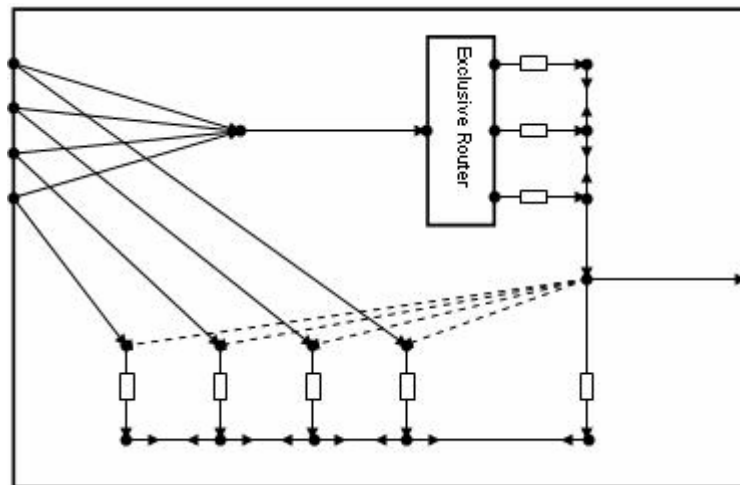


Figure 4-27 n out of m join in Reo

Chapter 5

Conclusion

In this chapter we present the conclusions of this thesis.

5.1. Summary and Conclusions

In this thesis, we assessed Reo and its control mechanisms as a workflow modeling language. Our goal was to show the Reo is powerful language that can be applied in various areas and workflow modeling has been chosen as one of them.

In order to achieve our goal, first, we specified the general definition of workflow control patterns in terms of some Point Interval Temporal Logic formulas. In the second step, we converted each PITL formula to a constraint automaton. In the third step, we implemented each workflow control pattern by a Reo circuit. In the fourth step, we compositionally derived the constraint automata of that Reo circuit and finally, in the fifth step, we showed the equivalence of the two constraint automata.

The reason that we used notion of time in our specifications and implementations was that in our opinion control flow can be best captured in terms of time since it is about ordering and synchronization that both have implicit semantics of time.

Following we briefly describe some of the advantages and useful mechanisms of Reo:

- Reo allows defining of new connectors out of primitive channels using composition rules. This makes Reo an adaptable language that can be well

suitable in different areas. As an example, we defined the `Delay` connector using `exclusiveRouter` connector and `sync` and `fifo1` primitive channels.

- Moreover, defining new connectors compositionally, each new connector also has well defined semantics that can be used for reasoning purposes.
- Reo provides a set of operations and patterns for the data used by operations.
- Reo connectors are defined as relations on timed data streams. Having the notion of time, it is possible to specify temporal constraints and verify Reo connectors.
- Besides the formal semantics, Reo provides graphical representations for its primitive channels and connectors, which makes it easier to use and understand.

Appendix A: Other Workflow Control Patterns

In this appendix we present Reo specification and implementation of structural, multiple instances, state-based, and cancellation patterns.

A.1. Specification and Implementation of Structural Patterns

In the following section, we present the specification and implementation of workflow structural pattern; Arbitrary Cycles pattern. We do not discuss Implicit Termination pattern here, since this pattern does not need any implementation at all.

A.1.1. Arbitrary Cycles

This pattern represents a situation in a workflow when one or more activities are executed repeatedly. As an example, in an insurance claim processing system, evaluating damage and producing report might be needed to be performed several times; sometimes the cycle should start over from evaluating the damage and sometimes it should start again from producing the report.

A.1.1.1. Workflow Arbitrary Cycles

Figure A-1 represents workflow Arbitrary Cycles pattern. In the workflow case of figure A-1, activity *B* may require to be executed multiple times.

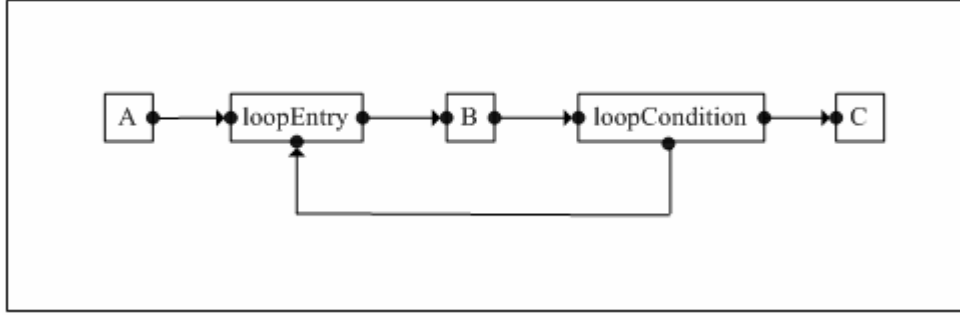


Figure A-5-1 Workflow Arbitrary Cycles

Formula (A-1) represents the PITL formula for this pattern.

$$\begin{aligned}
 & [(A \text{ Before } B) \vee (A \text{ Meets } B)] \wedge \\
 & \{ [(B \text{ Before } B) \vee (B \text{ Meets } B)] \vee [(B \text{ Before } C) \vee (B \text{ Meets } C)] \} \equiv \\
 & [(A_e < B_s \vee A_e = B_s)] \wedge [(B_e < B_s \vee B_e = B_s) \vee (B_e < C_s \vee B_e = C_s)] \equiv \\
 & (A_e \leq B_s) \wedge [(B_e \leq B_s) \vee (B_e \leq C_s)] \quad (A-1)
 \end{aligned}$$

In the above formula, activity *B* follows *A* and *C* follows *B*. After execution of activity *B*, it is possible to enable activity *C* or start again with activity *B*.

A.1.1.2. Reo Arbitrary Cycles

Figure A-2 represents the Reo implementation of Arbitrary Cycles pattern in Reo. In order to implement this pattern in Reo, we use Reo's XOR-Split and XOR-Join connectors.

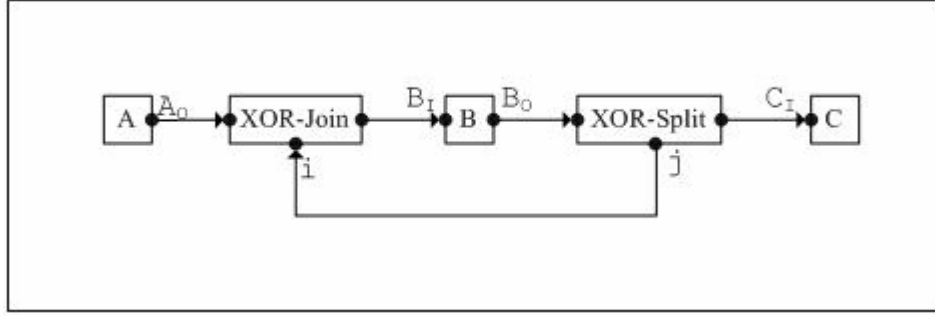


Figure A-5-2 Reo Arbitrary Cycle example

Now we present the TDS behavior of the Arbitrary Cycles connector:

1. For the XOR-Join connector consisting of nodes A_O , B_I , and i we have:

$$\begin{aligned} \text{XOR-Join}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(i), \tau(i) \rangle) &\equiv \\ \tau(A_{O0}) \neq \tau(i_0) \wedge & \\ \left\{ \begin{array}{l} d(A_{O0}) = d(B_{I0}) \wedge \tau(A_{O0}) \leq \tau(B_{I0}) \wedge \\ \text{XOR-Join}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(i_0), \tau(i_0) \rangle, \langle d(B_I)', \tau(B_I)' \rangle) \end{array} \right. & \text{if } \tau(A_{O0}) < \tau(i_0) \\ \left\{ \begin{array}{l} d(i_0) = d(B_{I0}) \wedge \tau(i_0) \leq \tau(B_{I0}) \wedge \\ \text{XOR-Join}(\langle d(A_O), \tau(A_O) \rangle, \langle d(i_0)', \tau(i_0)' \rangle, \langle d(B_I)', \tau(B_I)' \rangle) \end{array} \right. & \text{if } \tau(i_0) < \tau(A_{O0}) \end{aligned}$$

2. For the XOR-Split connector consisting of nodes B_O , C_I , and j we have:

$$\begin{aligned} \text{XOR-Split}(\langle d(C_O), \tau(C_O) \rangle, \langle d(D_I), \tau(D_I) \rangle, \langle d(j), \tau(j) \rangle) &\equiv \\ \left\{ \begin{array}{ll} d(C_O) = d(j) \wedge \tau(C_O) \leq \tau(j) & \text{if pat} \\ d(C_O) = d(D_I) \wedge \tau(C_O) \leq \tau(D_I) & \text{if } \neg\text{pat} \end{array} \right. & \end{aligned}$$

3. For the sync channel consisting of nodes i and j we have:

$$\langle d(j), \tau(j) \rangle_{\text{sync}} \langle d(i), \tau(i) \rangle \equiv d(j) = d(i) \wedge \tau(j) = \tau(i)$$

By composing the three formulas above, we obtain the TDS formula for Reo Arbitrary Cycles connector:

$$\begin{aligned}
& \text{c}_{\text{YC}}(\langle d(A_o), \tau(A_o) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \\
& \tau(A_{o0}) \neq \tau(i_0) \wedge \\
& \left\{ \begin{array}{l}
d(A_{o0}) = d(B_{I0}) \wedge \tau(A_{o0}) \leq \tau(B_{I0}) \wedge \\
\left\{ \begin{array}{l}
d(B_{o0}) = d(i_0) \wedge \tau(B_{o0}) \leq \tau(i_0) \quad \text{if pat} \\
d(B_{o0}) = d(C_{I0}) \wedge \tau(B_{o0}) \leq \tau(C_{I0}) \quad \text{if } \neg \text{pat}
\end{array} \right. \quad \text{if } \tau(i_0) < \tau(A_{o0}) \\
\text{c}_{\text{YC}}(\langle d(A_o)', \tau(A_o)' \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(C_I)', \tau(C_I)' \rangle) \\
d(i_0) = d(B_{I0}) \wedge \tau(i_0) \leq \tau(B_{o0}) \wedge \\
\left\{ \begin{array}{l}
d(B_{o0}) = d(i_0) \wedge \tau(B_{o0}) \leq \tau(i_0) \quad \text{if pat} \\
d(B_{o0}) = d(B_{I0}) \wedge \tau(B_{o0}) \leq \tau(C_{I0}) \quad \text{if } \neg \text{pat}
\end{array} \right. \quad \text{if } \tau(i_0) < \tau(A_{o0}) \\
\text{c}_{\text{YC}}(\langle d(A_o), \tau(A_o) \rangle, \langle d(B_I)', \tau(B_I)' \rangle, \langle d(C_I)', \tau(C_I)' \rangle)
\end{array} \right. \\
& \hspace{15em} (\text{A-2})
\end{aligned}$$

A.2. Specification and Implementation of Patterns Involving Multiple Instances

In the following section, we present the specification and implementation of workflow patterns that involve multiple instances.

A.2.1. Multiple Instances without Synchronization

This pattern represents a situation in a workflow when multiple instances of one activity should be created and executed in parallel. Each of these execution threads is independent of each other and there is no need to synchronize them. As an example, a customer ordering a book from an electronic bookstore such as Amazon may order multiple books at the same time. Many of the activities (e.g., billing, updating customer records, etc.) occur at the level of the order. However, within the order multiple instances need to be created to handle the activities related to one individual book (e.g., update stock levels, shipment, etc.). If the activities at the book level do not need to be synchronized, this pattern can be used.

A.2.1.1. Workflow Multiple Instances without Synchronization

Figure A-3 represents workflow Multiple Instances without Synchronization pattern in which n instances, b_1, \dots, b_n , of activity B are created. Then, activity C becomes enabled; the instances of activity B should not be synchronized in order for activity C to become enabled. It is also possible that no instance is created at all; in that case after completion of activity B , activity C will become enabled.

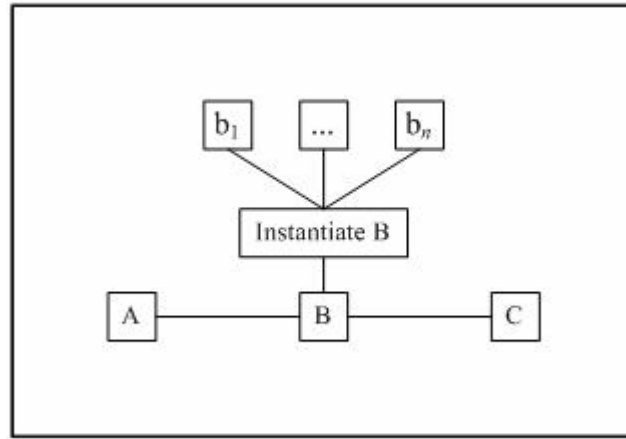


Figure A-5-3 Workflow Multiple Instances without Synchronization

Formula (A - 3) represents the PITL formula for this pattern.

$$\begin{aligned}
 & \left[(A \text{ Before } B) \vee (A \text{ Meets } B) \right] \wedge \left[(B \text{ Before } C) \vee (B \text{ Meets } C) \right] \wedge \\
 & \left[\left((B \text{ Overlaps } b_1) \wedge \dots \wedge (B \text{ Overlaps } b_n) \right) \vee \left((B \text{ During } b_1) \wedge \dots \wedge (B \text{ During } b_n) \right) \right] \equiv \\
 & \left[A_e < B_s \vee A_e = B_s \right] \wedge \left[B_e < C_s \vee B_e = C_s \right] \wedge \\
 & \left[\left(\forall i = 1..n : (B_s < b_{i_s} \wedge b_{i_s} < B_e \wedge B_e < b_{i_e}) \right) \vee \left(\forall i = 1..n : (B_s < b_{i_s} \wedge b_{i_e} < B_e) \right) \right] \equiv \\
 & A_e \leq B_s \wedge B_e \leq C_s \wedge (\forall = 1..n : B_s < b_{i_s}) \qquad (A-3)
 \end{aligned}$$

In the above formula, multiple instances are created while activity *B* has been enabled and will finish their execution either some time after activity *B* completes or before activity *B* finishes; but they are not synchronized.

A.2.1.2. Reo Multiple Instances without Synchronization

Figure A-4 represents the implementation of Multiple Instances without Synchronization pattern in Reo. In order to implement this pattern in Reo, we use Reo's AND-Split connector. After component instance *B* receives the execution control, it works as follows: (1) creates multiple instances by writing the token on its output node *B_{O1}*; (2) passes the execution control to activity *C* by writing on its output node *B_{O2}*, regardless of states of instances.

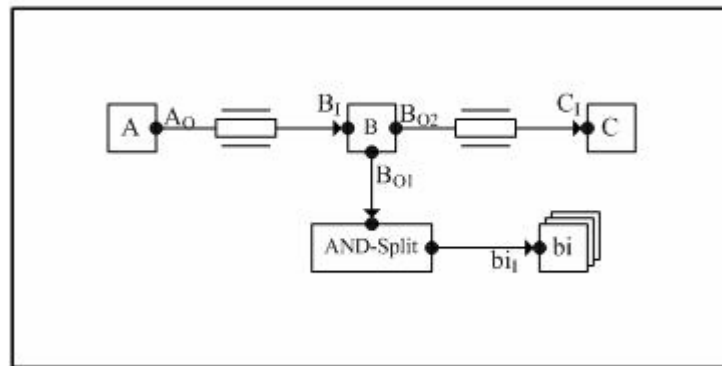


Figure A-5-4 Reo MI without synchronization connector

Now we present the TDS behavior of the above connector.

1. For the Delay connector between nodes *A_O* and *B_I* we have:

$$\langle d(A_O), \tau(A_O) \rangle_{\text{Delay}} \langle d(B_I), \tau(B_I) \rangle \equiv d(A_O) = d(B_I) \wedge \tau(A_O) \leq \tau(B_I)$$

2. For the AND-Split connector consists of nodes *B_{O1}* and *bi* we have:

$$\text{AND-Split}(\langle d(B_{o1}), \tau(B_{o1}) \rangle, \langle d(b1), \tau(b1) \rangle, \dots, \langle d(b1), \tau(b1) \rangle) \equiv \\ d(B_{o1}) = d(b1) = \dots = d(bn) \wedge \tau(B_{o1}) \leq \tau(b1) \wedge \dots \wedge \tau(B_{o1}) \leq \tau(bn)$$

3. For the Delay connector between nodes B_{o2} and C_l we have:

$$\langle d(B_{o2}), \tau(B_{o2}) \rangle_{\text{Delay}} \langle d(C_l), \tau(C_l) \rangle \equiv d(B_{o2}) = d(C_l) \wedge \tau(B_{o2}) \leq \tau(C_l)$$

By composing three formulas above, we obtain the TDS formula for Reo Multiple Instances without Synchronization:

$$\text{MI_1}(\langle d(A_o), \tau(A_o) \rangle, \langle d(B_l), \tau(B_l) \rangle, \langle d(B_{o1}), \tau(B_{o1}) \rangle, \langle d(B_{o2}), \tau(B_{o2}) \rangle \\ \langle d(b1_l), \tau(b1_l) \rangle, \dots, \langle d(bn), \tau(bn) \rangle, \langle d(C_l), \tau(C_l) \rangle) \equiv \\ (d(A_o) = d(B_l) \wedge \tau(A_o) \leq \tau(B_l)) \wedge (d(B_{o2}) = d(C_l) \wedge \tau(B_{o2}) \leq \tau(C_l)) \wedge \\ (\forall i = 1..n : d(B_{o1}) = d(bi_l) \wedge \tau(B_{o1}) \leq \tau(bi_l)) \quad (A-4)$$

Furthermore, we assume that $\tau(B_{o1}) < \tau(B_{o2})$; it means that multiple instances are created before activity B releases the token for activity C . Note that if no instances have to be created, activity B will only write the token to node B_{o2} .

A.2.2. Multiple Instances with Design Time Knowledge

This pattern represents a situation in a workflow when multiple instances of an activity should be created and executed in parallel. The number of instances required is known at design time. This pattern requires that multiple instances be synchronized before the next activity becomes activated. As an example, the requisition of hazardous material in a factory requires three different authorizations.

A.2.2.1. Workflow Multiple Instances with Design Time Knowledge

Figure A-5 represents workflow Multiple Instances with Design Time Knowledge pattern in which the number of required instances, n , is known at design time. In this pattern, when activity B becomes enabled, the required number of instances will be created, i.e. b_1, \dots, b_n , and after those instances finishes executing, activity C becomes enabled.

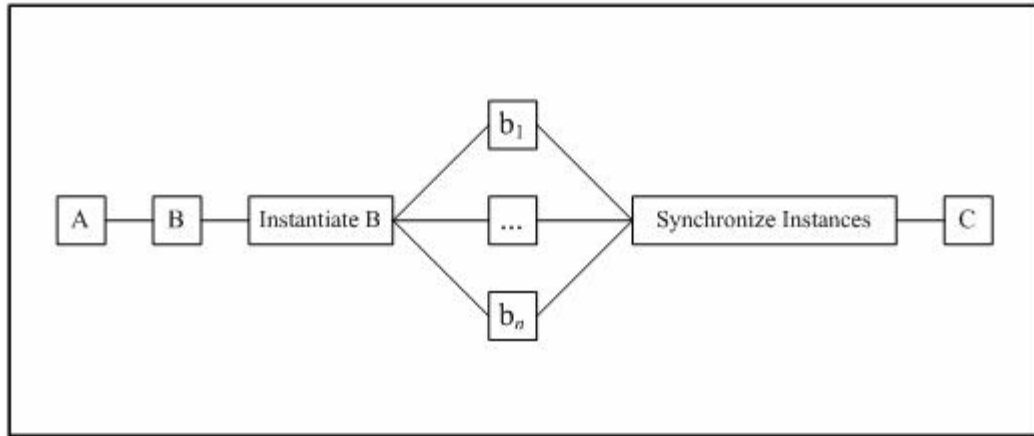


Figure A-5-5 Workflow Multiple Instances with Design Time Knowledge

Formula (A - 5) specifies the behaviour of this pattern.

$$\begin{aligned}
 & \left[(A \text{ Before } B) \vee (A \text{ Meets } B) \right] \wedge \\
 & \left[((B \text{ Before } b_1) \wedge \dots \wedge (B \text{ Before } b_n)) \vee ((B \text{ Meets } b_1) \wedge \dots \wedge (B \text{ Meets } b_n)) \right] \wedge \\
 & \left[((b_1 \text{ Before } C) \wedge \dots \wedge (b_n \text{ Before } C)) \vee ((b_1 \text{ Meets } C) \wedge \dots \wedge (b_n \text{ Meets } C)) \right] \equiv \\
 & \left[A_e < B_s \vee A_e = B_s \right] \wedge \left[\forall i = 1..n : (B_e < b_{i_s} \vee B_e = b_{i_s}) \right] \wedge \left[\forall i = 1..n : (b_{i_e} < C_s \vee b_{i_e} = C_s) \right] \equiv \\
 & A_e \leq B_s \wedge (\forall i = 1..n : (B_e \leq b_{i_s} \wedge b_{i_e} \leq C_s)) \quad (A-5)
 \end{aligned}$$

A.2.2.2. Reo Multiple Instances with Design Time Knowledge

Figure A-6 represents the Reo implementation of Multiple Instances without Synchronization pattern in Reo. Since the number of required instances is known at

design time, those instances are created using AND-Split and are synchronized using AND-Join.

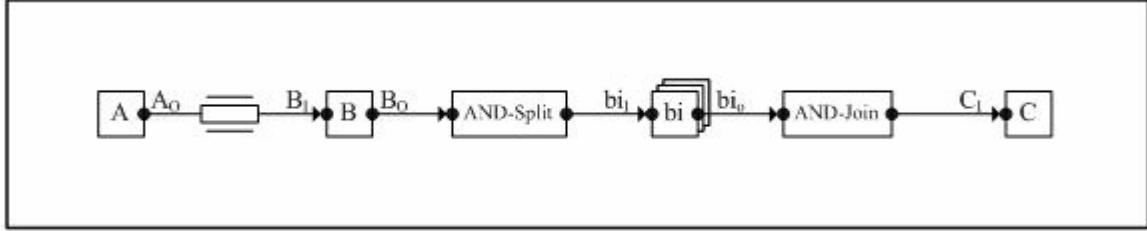


Figure A-5-6 Reo MI with Design Time Knowledge connector

Now we present the TDS behavior of the above connector.

1. For the Delay connector between nodes A_O and B_I we have:

$$\langle d(A_O), \tau(A_O) \rangle \text{Delay} \langle d(B_I), \tau(B_I) \rangle \equiv d(A_O) = d(B_I) \wedge \tau(A_O) \leq \tau(B_I)$$

2. For the AND-Split connector consisting of nodes B_O and bi_I we have:

$$\text{AND-Split}(\langle d(B_O), \tau(B_O) \rangle, \langle d(b_{1I}), \tau(b_{1I}) \rangle, \dots, \langle d(b_{nI}), \tau(b_{nI}) \rangle) \equiv d(B_O) = d(b_{1I}) = \dots = d(b_{nI}) \wedge \tau(B_O) \leq \tau(b_{1I}) \wedge \dots \wedge \tau(B_O) \leq \tau(b_{nI})$$

3. For the AND-Join connector consisting of nodes bi_O and C_I we have:

$$\text{AND-Join}(\langle d(b_{1O}), \tau(b_{1O}) \rangle, \dots, \langle d(b_{nO}), \tau(b_{nO}) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv (d(b_{1O}) = d(C_I) \wedge \tau(b_{1O}) \leq \tau(C_I)) \wedge \dots \wedge (d(b_{nO}) = d(C_I) \wedge \tau(b_{nO}) \leq \tau(C_I))$$

By composing three formulas above, we obtain the TDS formula for Reo Multiple Instances with Design Time Knowledge:

$$\begin{aligned}
& \text{MI_2}(\langle d(A_o), \tau(A_o) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(B_o), \tau(B_o) \rangle, \langle d(b1), \tau(b1) \rangle, \dots, \\
& \quad \langle d(bn), \tau(bn) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \\
& (d(A_o) = d(B_I) \wedge \tau(A_o) \leq \tau(B_I)) \wedge (\forall i = 1..n : d(B_o) = d(bi_I) \wedge \tau(B_o) \leq \tau(bi_I)) \wedge \\
& (\forall i = 1..n : d(bi_o) = d(C_I) \wedge \tau(bi_o) \leq \tau(C_I)) \tag{A-6}
\end{aligned}$$

A.2.3. Multiple Instances with Run Time Knowledge

This pattern represents a situation in a workflow when multiple instances of an activity should be created and executed in parallel. The number of instances required is known at run time and before the time instances have to be actually created. This pattern requires that multiple instances be synchronized before the next activity becomes activated. As an example, in the review process of a scientific paper submitted to a journal, reviewing a paper is instantiated several times depending on the content of the paper, the availability of referees, and the credentials of the authors. Only if all reviews have been returned, processing is continued.

A.2.3.1. Workflow Multiple Instances with Run Time Knowledge

Figure A-7 represents workflow Multiple Instances with Run Time Knowledge pattern in which the number of required instances, n , is known at run time.

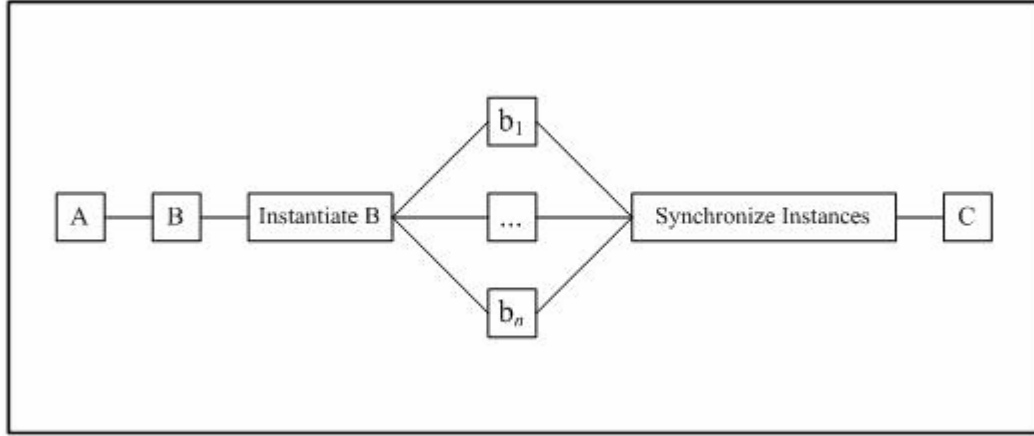


Figure A-5-7 Workflow Multiple Instances with Run Time Knowledge

Formula (A - 7) specifies the behaviour of this pattern.

$$\begin{aligned}
 & [(A \text{ Before } B) \vee (A \text{ Meets } B)] \wedge \\
 & [((B \text{ Before } b_1) \wedge \dots \wedge (B \text{ Before } b_n)) \vee ((B \text{ Meets } b_1) \wedge \dots \wedge (B \text{ Meets } b_n))] \wedge \\
 & [((b_1 \text{ Before } C) \wedge \dots \wedge (b_n \text{ Before } C)) \vee ((b_1 \text{ Meets } C) \wedge \dots \wedge (b_n \text{ Meets } C))] \equiv \\
 & [A_e < B_s \vee A_e = B_s] \wedge [\forall i = 1..n : (B_e < b_{i_s} \vee B_e = b_{i_s})] \wedge [\forall i = 1..n : (b_{i_e} < C_s \vee b_{i_e} = C_s)] \equiv \\
 & A_e \leq B_s \wedge (\forall i = 1..n : (B_e \leq b_{i_s} \wedge b_{i_e} \leq C_s)) \quad (A-7)
 \end{aligned}$$

A.2.3.2. Reo Multiple Instances with Run Time Knowledge

Figure A-8 represents the Reo implementation of Multiple Instances with Run Time Knowledge pattern in Reo. We implement this pattern as follows. The multiple instances are created using an AND-Split connector. For synchronizing the instances we introduce a new helper activity *BI* and use the Multi Merge connector. After each instance of activity *B* finishes, activity *BI* becomes enabled once and decides whether all instances have finished their execution or not. If all have finished, *BI* will enabled activity *C*; hence synchronize the instances. If they have not finished yet, it waits for other instances to complete.

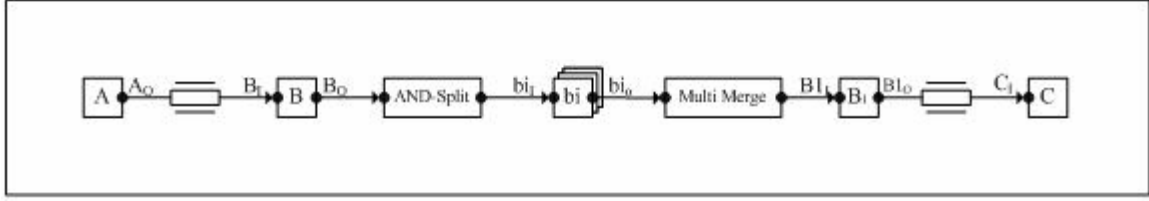


Figure A-5-8 Reo MI with Run Time Knowledge connector

Now we present the TDS behavior of the Multiple Instances with Run Time Knowledge connector.

1. For the Delay connector between nodes A_O and B_I we have:

$$\langle d(A_O), \tau(A_O) \rangle \text{Delay} \langle d(B_I), \tau(B_I) \rangle \equiv d(A_O) = d(B_I) \wedge \tau(A_O) \leq \tau(B_I)$$

2. For the AND-Split connector consisting of nodes B_O and bi_I we have:

$$\text{AND-Split}(\langle d(B_O), \tau(B_O) \rangle, \langle d(b_{1I}), \tau(b_{1I}) \rangle, \dots, \langle d(b_{nI}), \tau(b_{nI}) \rangle) \equiv \\ d(B_O) = d(b_{1I}) = \dots = d(b_{nI}) \wedge \tau(B_O) \leq \tau(b_{1I}) \wedge \dots \wedge \tau(B_O) \leq \tau(b_{nI})$$

3. For the Multi Merge connector consisting of nodes bi_O and B_I we have:

$$\text{MultiMerge}(\langle d(b_{1O}), \tau(b_{1O}) \rangle, \dots, \langle d(b_{nO}), \tau(b_{nO}) \rangle, \langle d(B_I), \tau(B_I) \rangle) \equiv \\ \forall i = 1..n, j = 1..n : \tau(b_{iO}) \neq \tau(b_{jO}) \wedge \\ d(b_{iO}) = d(B_I) \wedge \tau(b_{iO}) \leq \tau(B_I) \wedge \\ \text{MultiMerge}(\langle d(b_{1O}), \tau(b_{1O}) \rangle, \langle d(b_{iO})', \tau(b_{iO})' \rangle, \quad \text{if } \tau(b_{iO}) < \tau(b_{jO}) \\ \dots, \langle d(b_{jO}), \tau(b_{jO}) \rangle, \dots, \langle d(b_{nO}), \tau(b_{nO}) \rangle, \langle d(B_I)', \tau(B_I)' \rangle)$$

4. For the Delay connector between nodes B_I_O and C_I we have:

$$\langle d(B_I_O), \tau(B_I_O) \rangle \text{Delay} \langle d(C_I), \tau(C_I) \rangle \equiv \\ d(B_I_O) = d(C_I) \wedge \tau(B_I_O) \leq \tau(C_I)$$

By composing three formulas above, we obtain the TDS formula for Reo Multiple Instances with Run Time Knowledge:

$$\begin{aligned}
& \text{MI}_{\exists} \left(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(B_{I1}), \tau(B_{I1}) \rangle, \langle d(B_{I0}), \tau(B_{I0}) \rangle, \right. \\
& \quad \left. \langle d(b1), \tau(b1) \rangle, \dots, \langle d(bn), \tau(bn) \rangle, d(C_I), \tau(C_I) \right) \equiv \\
& \forall i = 1..n, j = 1..n : \tau(bi_{I0}) \neq \tau(bj_{I0}) \wedge \\
& d(A_{O0}) = d(B_{I0}) \wedge d(B_{O0}) = d(bi_{I0}) \wedge d(bi_{O0}) = d(B_{I10}) \wedge d(B_{I00}) = d(C_{I0}) \wedge \\
& \tau(A_O) \leq \tau(B_I) < \tau(B_O) \leq \tau(bi_{I0}) < \tau(bi_{O0}) \leq \tau(B_{I10}) < \tau(B_{I00}) \leq \tau(C_{I0}) \quad \text{if } \tau(bi_{I0}) < \tau(bj_{I0}) \\
& \text{MI}_{\exists} \left(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(B_I)', \tau(B_I)' \rangle, \langle d(B_O)', \tau(B_O)' \rangle, \langle d(B_{I1})', \tau(B_{I1})' \rangle, \langle d(B_{I0})', \tau(B_{I0})' \rangle, \right. \\
& \quad \left. \langle d(b1), \tau(b1) \rangle, \dots, \langle d(bi)', \tau(bi)' \rangle, \dots, \langle d(bn), \tau(bn) \rangle, d(C_I), \tau(C_I) \right) \quad (\text{A-8})
\end{aligned}$$

A.2.4. Multiple Instances without Run Time Knowledge

This pattern represents a situation in a workflow when multiple instances of an activity should be created and executed in parallel. The number of instances required is not known at run time and even before the time instances have to actually be created. Thus, it is possible that when some of the instances are being executed and some of them are completed, more instances need to be created and executed in parallel. In addition, this pattern requires that multiple instances be synchronized before the next activity becomes activated. As an example, for the processing of an insurance claim, zero or more eyewitness reports should be handled. The number of eyewitness reports may vary. Even when processing eyewitness reports for a given insurance claim, new eyewitnesses may surface and the number of instances may change.

A.2.4.1. Workflow Multiple Instances without Run Time Knowledge

Figure A-9 represents workflow Multiple Instances without Run Time Knowledge.

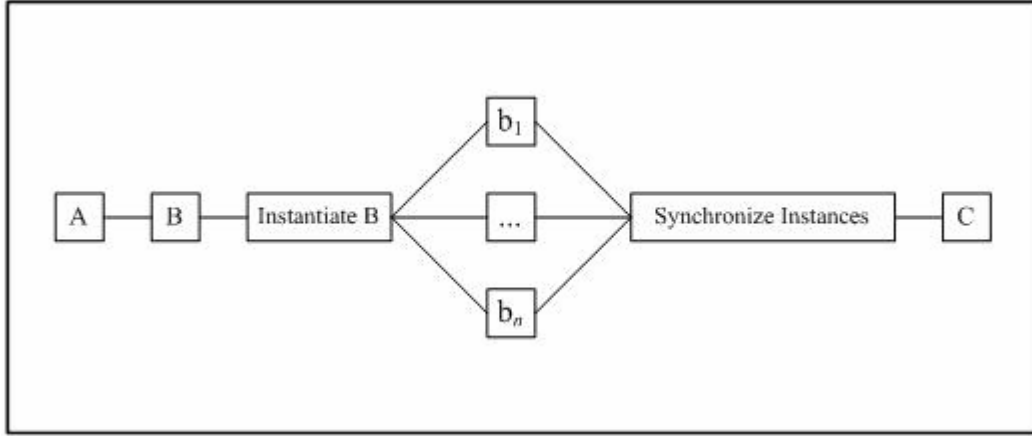


Figure A-5-9 Workflow Multiple Instances without Run Time Knowledge

Formula (A - 9) specifies the behaviour of this pattern.

$$\begin{aligned}
 & \left[(A \text{ Before } B) \vee (A \text{ Meets } B) \right] \wedge \\
 & \left[\forall i = 1..n : ((B \text{ Before } bi) \vee (B \text{ Meets } bi)) \right] \wedge \left[\forall i = 1..n : ((bi \text{ Before } C) \vee (bi \text{ Meets } C)) \right] \equiv \\
 & \left[A_e < B_s \vee A_e = B_s \right] \wedge \left[\forall i = 1..n : (B_e < b_{i_s} \vee B_e = b_{i_s}) \right] \wedge \left[\forall i = 1..n : (b_{i_e} < C_s \vee b_{i_e} = C_s) \right] \equiv \\
 & A_e \leq B_s \wedge (\forall i = 1..n : B_e \leq b_{i_s} \wedge b_{i_e} \leq C_s) \wedge (\forall i = 1..n, j = 1..n : (b_{i_s} \leq b_{j_s} \vee b_{i_s} \geq b_{j_s})) \quad (A-9)
 \end{aligned}$$

A.2.4.2. Reo Multiple Instances without Run Time Knowledge

Figure A-10 represents the Reo implementation of Multiple Instances without Run Time Knowledge in Reo. We implement this pattern using AND-Split, Multi Merge, XOR-Split, and XOR-Join connectors. After the completion of activity A, activity B becomes enabled and creates the required instances using the AND-Split connector. We introduce a helper activity, *BI*, which is, as in previous pattern, responsible for synchronizing the multiple instances. Activity *BI* works as follows: whenever any of the instances finishes executing, according to Multi Merge connector, *BI* becomes enabled; it checks whether new instances should be created or not. If new instances are required, by using the XOR-Split connector, activity *B* becomes enabled again and creates more instances. If new instances are not required, *BI* will wait for all instances to be

completed and synchronize them. Then, it will enable activity *C* using XOR-Split Connector.

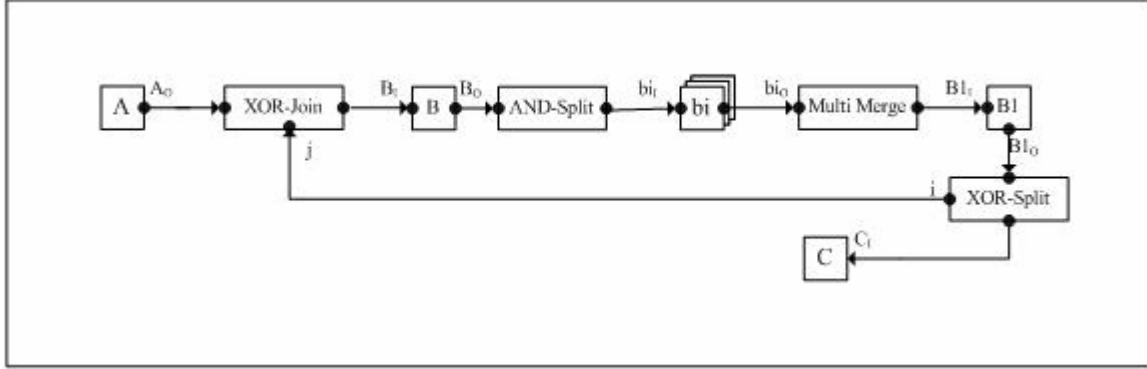


Figure A-5-10 Reo MI without Run Time Knowledge connector

Now we present the TDS behavior of the Multiple Instances without Run Time Knowledge connector.

1. For the XOR-Join connector consists of nodes A_O, j , and B_I we have:

$$\begin{aligned} \text{XOR-Join}(\langle d(A_O), \tau(A_O) \rangle, \langle d(j), \tau(j) \rangle, \langle d(B_I), \tau(B_I) \rangle) &\equiv \\ \tau(A_{O0}) \neq \tau(j_0) \wedge & \\ \begin{cases} d(A_{O0}) = d(B_{I0}) \wedge \tau(A_{O0}) \leq \tau(B_{I0}) \wedge \\ \text{XOR-Join}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(j_0), \tau(j_0) \rangle, \langle d(B_{I0})', \tau(B_{I0})' \rangle) & \text{if } \tau(A_{O0}) < \tau(j_0) \\ d(j_0) = d(B_{I0}) \wedge \tau(j_0) \leq \tau(B_{I0}) \wedge \\ \text{XOR-Join}(\langle d(A_O), \tau(A_O) \rangle, \langle d(j_0)', \tau(j_0)' \rangle, \langle d(B_{I0})', \tau(B_{I0})' \rangle) & \text{if } \tau(j_0) < \tau(A_{O0}) \end{cases} & \end{aligned}$$

2. For the AND-Split connector consists of nodes B_O and bi_I we have:

$$\begin{aligned} \text{AND-Split}(\langle d(B_O), \tau(B_O) \rangle, \langle d(b_{I1}), \tau(b_{I1}) \rangle, \dots, \langle d(b_{In}), \tau(b_{In}) \rangle) &\equiv \\ d(B_O) = d(b_{I1}) = \dots = d(b_{In}) \wedge \tau(B_O) \leq \tau(b_{I1}) \wedge \dots \wedge \tau(B_O) \leq \tau(b_{In}) & \end{aligned}$$

3. For the Multi Merge connector consists of nodes bi_O and B_I we have:

$$\begin{aligned}
& \text{MultiMerge}(\langle d(b1_O), \tau(b1_O) \rangle, \dots, \langle d(bn_O), \tau(bn_O) \rangle, \langle d(B1_I), \tau(B1_I) \rangle) \equiv \\
& \forall i = 1..n, j = 1..n : \tau(bi_{O0}) \neq \tau(bj_{O0}) \wedge \\
& d(bi_{O0}) = d(B1_I) \wedge \tau(bi_{O0}) \leq \tau(B1_I) \wedge \\
& \text{MultiMerge} \left(\langle d(b1_O), \tau(b1_O) \rangle, \langle d(bi_O)', \tau(bi_O)' \rangle, \quad \text{if } \tau(bi_{O0}) < \tau(bj_{O0}) \right. \\
& \quad \left. \dots, \langle d(bj_O), \tau(bj_O) \rangle, \dots, \langle d(bn_O), \tau(bn_O) \rangle, \langle d(B1_I)', \tau(B1_I)' \rangle \right)
\end{aligned}$$

4. For the XOR-Split connector consists of nodes $B1_O$, i and C_I we have:

$$\begin{aligned}
& \text{XOR-Split}(\langle d(B1_O), \tau(B1_O) \rangle, \langle d(i), \tau(i) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \\
& \begin{cases} d(B1_O) = d(i) \wedge \tau(B1_O) \leq \tau(i) & \text{if pat} \\ d(B1_O) = d(C_I) \wedge \tau(A_O) \leq \tau(C_I) & \text{if } \neg \text{pat} \end{cases}
\end{aligned}$$

By composing four formulas above, we obtain the TDS formula for Reo Multiple Instances with Run Time Knowledge:

$$\begin{aligned}
& \text{MI_4}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(B1_I), \tau(B1_I) \rangle, \langle d(B1_O), \tau(B1_O) \rangle, \\
& \quad \langle d(b1), \tau(b1) \rangle, \dots, \langle d(bn), \tau(bn) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \forall i = 1..n, j = 1..n : \tau(A_{O0}) \neq \tau(j_{O0}) \\
& \left\{ \begin{array}{l} d(A_{O0}) = d(B1_O) \wedge d(B_{O0}) = d(bi_{I0}) \wedge d(bi_{O0}) = d(B1_I0) \wedge \\ \tau(A_{O0}) \leq \tau(B1_O) < \tau(B_{O0}) \leq \tau(bi_{I0}) < \tau(bi_{O0}) \leq \tau(B1_I0) < \tau(B1_{O0}) \wedge \\ \tau(bi_{O0}) \neq \tau(bj_{O0}) \wedge \begin{cases} d(B1_{O0}) = d(j_{O0}) \wedge \tau(B1_{O0}) \leq \tau(j_{O0}) & \text{if pat} \\ d(B1_{O0}) = d(C_{I0}) \wedge \tau(B1_{O0}) \leq \tau(C_{I0}) & \text{if } \neg \text{pat} \end{cases} \end{array} \right. \\
& \left. \begin{array}{l} d(j_{O0}) = d(B1_O) \wedge d(B_{O0}) = d(bi_{I0}) \wedge d(bi_{O0}) = d(B1_I0) \wedge \\ \tau(j_{O0}) \leq \tau(B1_O) < \tau(B_{O0}) \leq \tau(bi_{I0}) < \tau(bi_{O0}) \leq \tau(B1_I0) < \tau(B1_{O0}) \wedge \quad \text{if } \tau(j_{O0}) < \tau(A_{O0}) \\ \tau(bi_{O0}) \neq \tau(bj_{O0}) \wedge \begin{cases} d(B1_{O0}) = d(j_{O0}) \wedge \tau(B1_{O0}) \leq \tau(j_{O0}) & \text{if pat} \\ d(B1_{O0}) = d(C_{I0}) \wedge \tau(B1_{O0}) \leq \tau(C_{I0}) & \text{if } \neg \text{pat} \end{cases} \end{array} \right. \\
& \wedge \text{MI_4}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(B_I)', \tau(B_I)' \rangle, \langle d(B_O)', \tau(B_O)' \rangle, \langle d(B1_I)', \tau(B1_I)' \rangle, \langle d(B1_O)', \tau(B1_O)' \rangle, \\
& \quad \langle d(b1), \tau(b1) \rangle, \dots, \langle d(bi)', \tau(bi)' \rangle, \dots, \langle d(bn), \tau(bn) \rangle, \langle d(C_I), \tau(C_I) \rangle) \quad (\text{A-10})
\end{aligned}$$

Note that whenever the pattern `pat` is true, then more instances should be created.

A.3. State-based Patterns

In the following section, we present the specification and implementation of workflow state-based patterns.

A.3.1. Deferred Choice

This pattern represents a situation in a workflow when, among several alternative branches, one is chosen and the others are withdrawn. The difference with XOR-Split pattern is that the choice is not made explicitly (based on some conditions or control data); some alternatives are offered to the environment and the environment activates one of the branches (e.g. sending a signal to the activity). It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible. This pattern is also known as *Deferred XOR-Split*. As an example, in an organization, after receiving products there are two ways to transport them to the destination department. The selection is based on the availability of the corresponding resources. Therefore, the choice is deferred until a resource is available.

A.3.1.1. Workflow Deferred Choice

Figure A-11 represents workflow Deferred Choice pattern in which after completion of activity *A*, based on the signal received from the environment of the workflow case, *signal1* or *signal2*, either activity *B* or *C* becomes enabled, respectively.

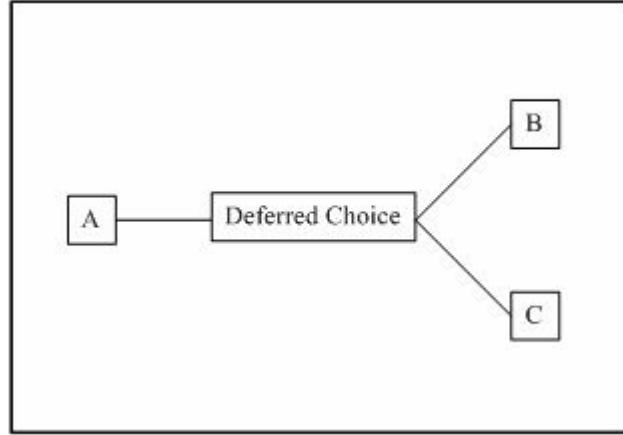


Figure A-5-11 Workflow Deferred Choice

Formula (A -11) represents the PITL formula for this pattern.

$$\begin{aligned}
 & \left\{ \begin{array}{l} (signal1 \text{ Starts } A) \rightarrow (A \text{ Before } B) \vee (A \text{ Meets } B) \\ (signal2 \text{ Starts } A) \rightarrow (A \text{ Before } C) \vee (A \text{ Meets } C) \end{array} \right. \equiv \\
 & \left\{ \begin{array}{l} A_e = signal1 \rightarrow (A_e < B_s \vee A_e = B_s) \\ A_e = signal2 \rightarrow (A_e < C_s \vee A_e = C_s) \end{array} \right. \equiv \left\{ \begin{array}{l} A_e = signal1 \rightarrow A_e \leq B_s \\ A_e = signal2 \rightarrow A_e \leq C_s \end{array} \right. \quad (A-11)
 \end{aligned}$$

The above formula states that if at the time activity *A* is completed *signal1* is received, then activity *B* follows; but if at the time activity *A* is completed *signal2* is received, then activity *C* follows *A*. In the above formula, *signal1* and *signal2* are zero-length intervals at which *signal1* and *signal2* are received, respectively.

A.3.1.2. Reo Deferred Choice

Figure A-12 represents the implementation of Deferred Choice pattern in Reo. In order to implement this pattern, we model the environment as an activity called *Environment*; this activity is responsible for receiving the signals from the actual environment. After the execution of *A*, control transfers to *Environment*. Then, it decides which one of activities *B* or *C* should be executed next. According to this decision, the XOR-Split connector enables either *B* or *C*.

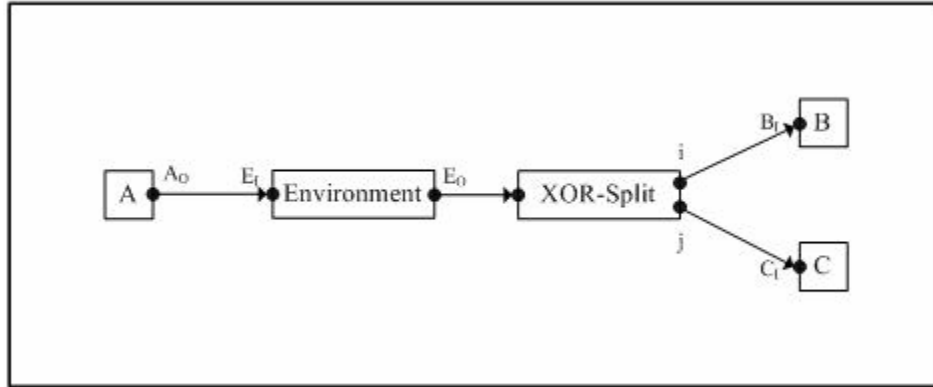


Figure A-5-12 Reo Deferred Choice connector

The TDS formula for this pattern is the same as formula (4-11) for XOR-Split connector. The only difference here is that the conditions in Deferred Choice connector are actually signals from the environment. Formula (A-12) represents the TDS formula for this connector.

$$\text{DeferredChoice}(\langle d(E_o), \tau(E_o) \rangle, \langle d(B_i), \tau(B_i) \rangle, \langle d(C_i), \tau(C_i) \rangle) \equiv \begin{cases} d(E_o) = d(B_i) \wedge \tau(E_o) \leq \tau(B_i) & \text{if signal1} \\ d(E_o) = d(C_i) \wedge \tau(E_o) \leq \tau(C_i) & \text{if signal2} \end{cases} \quad (\text{A-12})$$

A.3.2. Interleaved Parallel Routing

This pattern represents a situation in a workflow when several activities have to be executed in an arbitrary order. The order of execution is determined at run time for each workflow instance. It is assumed that no two activities are executed in parallel. This activity is also known as *Unordered Sequence*. As an example, the Navy requires every job applicant to take two tests: physical test and mental test. These tests can be conducted in any order but not at the same time.

A.3.2.1. Workflow Interleaved Parallel Routing

Figure A-13 represents workflow Interleaved Parallel Routing pattern in which after completion of activity A, activities B and C become enabled after each other in an

arbitrary order; the fact that which activity should be enabled first depends on some condition at run time.

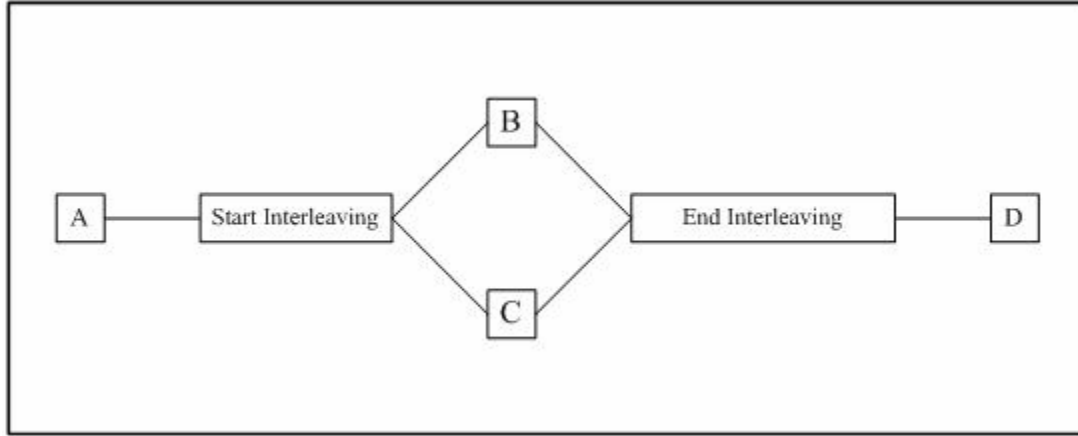


Figure A-5-13 Workflow Interleaved Parallel Routing

Formula (A - 13) represents the PITL formula for this pattern.

$$\begin{aligned}
 & \left[\left((A \text{ Before } B) \vee (A \text{ Meets } B) \right) \wedge \left((B \text{ Before } C) \vee (B \text{ Meets } C) \right) \wedge \right. \\
 & \left. \left((C \text{ Before } D) \vee (C \text{ Meets } D) \right) \right] \vee \\
 & \left[\left((A \text{ Before } C) \vee (A \text{ Meets } C) \right) \wedge \left((C \text{ Before } B) \vee (C \text{ Meets } B) \right) \wedge \right. \\
 & \left. \left((B \text{ Before } D) \vee (B \text{ Meets } D) \right) \right] \equiv \\
 & \left[(A_e < B_s \vee A_e = B_s) \wedge (B_e < C_s \vee B_e = C_s) \wedge (C_e < D_s \vee C_e = D_s) \right] \vee \\
 & \left[(A_e < C_s \vee A_e = C_s) \wedge (C_e < B_s \vee C_e = B_s) \wedge (B_e < D_s \vee B_e = D_s) \right] \equiv \\
 & (A_e \leq B_s \wedge B_e \leq C_s \wedge C_e \leq D_s) \vee (A_e \leq C_s \wedge C_e \leq B_s \wedge B_e \leq D_s) \quad (A-13)
 \end{aligned}$$

The above formula states that if the completion of activity *A* happens either some time before or at the same time activity *B* becomes enabled (which means activity *B* is executed before activity *C*), then, completion of *B* happens some time before or at the same time activity *C* becomes enabled and completion of *C* happens some time before or at the same time activity *D* becomes enabled. But if the completion of activity *A* happens either some time before or at the same time activity *C* becomes enabled (which means

activity *B* is executed before activity *B*), then, completion of *C* happens some time before or at the same time activity *B* becomes enabled and completion of *B* happens some time before or at the same time activity *D* becomes enabled.

A.3.2.2. Reo Interleaved Parallel Routing

Figure A-14 represents the Reo implementation of Interleaved Parallel Routing in Reo. We model this pattern in the context of a loop. After execution of *A*, one of the activities *B* or *C* will be executed according to some condition at run time (the decision is made by the Deferred Choice connector). Assume that *B* becomes enabled first. After *B* finishes executing, the XOR-Split connector checks whether all activities, i.e. here *B* and *C*, have been executed or not. If not, the loop continues with the Deferred Choice connector and activity *C* becomes enabled. After *C* finishes executing, activity *D* will become enabled. If there are more than two activities, then among the rest of activities, one is chosen in Deferred Choice and the flow will continue until all activities are executed in an arbitrary order.

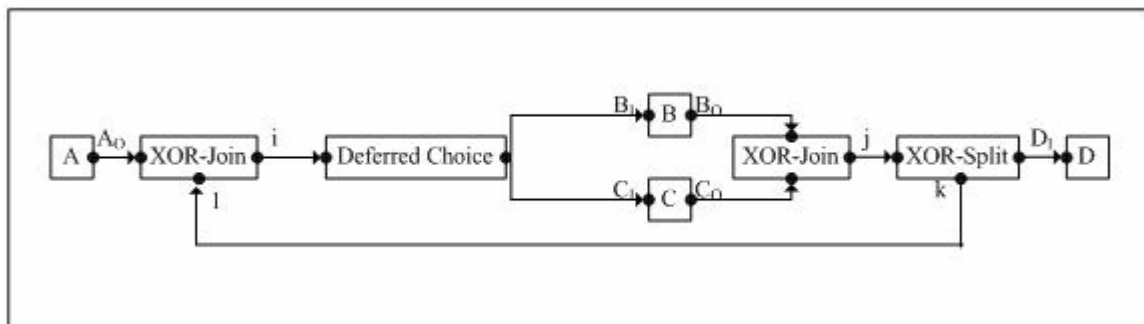


Figure A-5-14 Reo Interleaved Parallel Routing connector

Now we present the TDS behavior of the Interleaved Parallel Routing connector.

1. For the XOR-Join connector consisting of nodes A_0 , l , and i we have:

$$\begin{aligned}
& \text{XOR-Join}(\langle d(A_O), \tau(A_O) \rangle, \langle d(l), \tau(l) \rangle, \langle d(i), \tau(i) \rangle) \equiv \\
& \tau(A_{O0}) \neq \tau(l_0) \wedge \\
& \left\{ \begin{array}{l} d(A_{O0}) = d(i_0) \wedge \tau(A_{O0}) \leq \tau(i_0) \wedge \\ \text{XOR-Join}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(l), \tau(l) \rangle, \langle d(i)', \tau(i)' \rangle) \quad \text{if } \tau(A_{O0}) < \tau(l_0) \\ d(l_0) = d(i_0) \wedge \tau(l_0) \leq \tau(i_0) \wedge \\ \text{XOR-Join}(\langle d(A_O), \tau(A_O) \rangle, \langle d(l)', \tau(l)' \rangle, \langle d(i)', \tau(i)' \rangle) \quad \text{if } \tau(l_0) < \tau(A_{O0}) \end{array} \right.
\end{aligned}$$

2. For the Deferred Choice connector consisting of nodes i , B_I and C_I we have:

$$\begin{aligned}
& \text{DeferredChoice}(\langle d(i), \tau(i) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \\
& \left\{ \begin{array}{l} d(i) = d(B_I) \wedge \tau(i) \leq \tau(B_I) \quad \text{if } \text{signal1} \\ d(i) = d(C_I) \wedge \tau(i) \leq \tau(C_I) \quad \text{if } \text{signal2} \end{array} \right.
\end{aligned}$$

3. For the XOR-Join connector consisting of nodes B_O , C_O , and j we have:

$$\begin{aligned}
& \text{XOR-Join}(\langle d(B_O), \tau(B_O) \rangle, \langle d(C_O), \tau(C_O) \rangle, \langle d(j), \tau(j) \rangle) \equiv \\
& \tau(B_{O0}) \neq \tau(C_{O0}) \wedge \\
& \left\{ \begin{array}{l} d(B_{O0}) = d(j_0) \wedge \tau(B_{O0}) \leq \tau(j_0) \wedge \\ \text{XOR-Join}(\langle d(B_O)', \tau(B_O)' \rangle, \langle d(C_O), \tau(C_O) \rangle, \langle d(j)', \tau(j)' \rangle) \quad \text{if } \tau(B_{O0}) < \tau(C_{O0}) \\ d(C_{O0}) = d(j_0) \wedge \tau(C_{O0}) \leq \tau(j_0) \wedge \\ \text{XOR-Join}(\langle d(B_O), \tau(B_O) \rangle, \langle d(C_O)', \tau(C_O)' \rangle, \langle d(j)', \tau(j)' \rangle) \quad \text{if } \tau(C_{O0}) < \tau(B_{O0}) \end{array} \right.
\end{aligned}$$

4. For the XOR-Split connector consists of nodes j , k , and D_I we have:

$$\begin{aligned}
& \text{XOR-Split}(\langle d(j), \tau(j) \rangle, \langle d(k), \tau(k) \rangle, \langle d(D_I), \tau(D_I) \rangle) \equiv \\
& \left\{ \begin{array}{l} d(j) = d(k) \wedge \tau(j) \leq \tau(k) \quad \text{if another activity has to be enabled} \\ d(j) = d(D_I) \wedge \tau(j) \leq \tau(D_I) \quad \text{if both activities have been completed} \end{array} \right.
\end{aligned}$$

By composing four formulas above, we obtain the TDS formula for Reo Interleaved Parallel Routing:

$$\begin{aligned} & \text{IPR}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(C_I), \tau(C_I) \rangle, \\ & \quad \langle d(B_O), \tau(B_O) \rangle, \langle d(C_O), \tau(C_O) \rangle, \langle d(D_I), \tau(D_I) \rangle) \equiv \end{aligned} \quad (A-14)$$

$$\tau(A_{O0}) \neq \tau(l_0) \wedge \tau(B_{O0}) \neq \tau(C_{O0}) \wedge$$

$$\begin{cases} d(A_{O0}) = d(B_{I0}) \wedge \tau(A_{O0}) = \tau(B_{I0}) & \text{if } \tau(A_{O0}) \leq \tau(l_0) \wedge \text{milestone} \\ d(A_{O0}) = d(C_{I0}) \wedge \tau(A_{O0}) = \tau(C_{I0}) & \text{if } \tau(A_{O0}) \leq \tau(l_0) \wedge \neg \text{milestone} \end{cases} \wedge$$

$$\begin{cases} d(B_{O0}) = d(D_{I0}) \wedge \tau(B_{O0}) \leq \tau(D_{I0}) \wedge \text{IPR}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(B_I)', \tau(B_I)' \rangle, \langle d(C_I), \tau(C_I) \rangle, \\ \langle d(B_O)', \tau(B_O)' \rangle, \langle d(C_O), \tau(C_O) \rangle, \langle d(D_I)', \tau(D_I)' \rangle) & \text{if } \tau(B_{O0}) \leq \tau(C_{O0}) \wedge \text{no more activity} \\ \\ d(B_{O0}) = d(l_0) \wedge \tau(B_{O0}) \leq \tau(l_0) & \text{if } \tau(B_{O0}) \leq \tau(C_{O0}) \wedge \text{more activities} \\ \\ d(C_{O0}) = d(D_{I0}) \wedge \tau(C_{O0}) \leq \tau(D_{I0}) \wedge \text{IPR}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(C_I)', \tau(C_I)' \rangle, \\ \langle d(B_O), \tau(B_O) \rangle, \langle d(C_O)', \tau(C_O)' \rangle, \langle d(D_I)', \tau(D_I)' \rangle) & \text{if } \tau(C_{O0}) \leq \tau(B_{O0}) \wedge \text{no more activity} \\ \\ d(C_{O0}) = d(l_0) \wedge \tau(C_{O0}) \leq \tau(l_0) & \text{if } \tau(C_{O0}) \leq \tau(B_{O0}) \wedge \neg \text{more activities} \end{cases}$$

A.3.3 Milestone

This pattern represents a situation in a workflow when enabling of an activity depends on the case being in a specified state, i.e. the activity is only enabled if a certain milestone has been reached which did not yet expire. As an example, in a travel agency, flights, rental cars, and hotels may be booked as long as the invoice is not printed.

A.3.2.1. Workflow Milestone

Figure A-15 represents workflow Milestone pattern in which after completion of activity A, based on a milestone, *milestone*, either activity B or C becomes enabled, respectively.

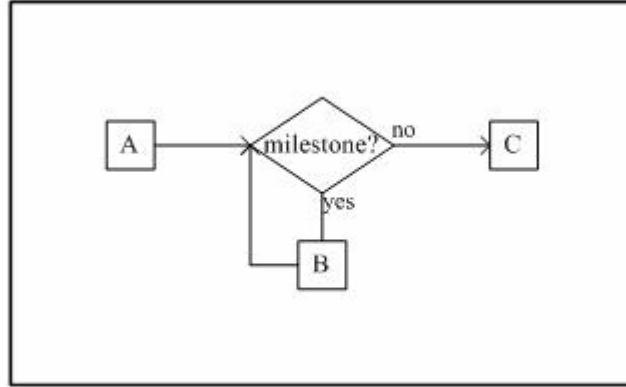


Figure A-5-15 Workflow Milestone

Formula (A - 15) represents the PITL formula for this pattern.

$$\begin{aligned}
& \left[(\text{milestone } \mathbf{Finishes} A) \rightarrow (A \mathbf{Before} B) \vee (A \mathbf{Meets} B) \right] \wedge \\
& \left[(\neg \text{milestone } \mathbf{Finishes} A) \rightarrow (A \mathbf{Before} C) \vee (A \mathbf{Meets} C) \right] \wedge \\
& \left[(\text{milestone } \mathbf{Finishes} B) \rightarrow (B \mathbf{Before} B) \vee (B \mathbf{Meets} B) \right] \wedge \\
& \left[(\neg \text{milestone } \mathbf{Finishes} B) \rightarrow (B \mathbf{Before} C) \vee (B \mathbf{Meets} C) \right] \equiv \\
& \left[A_e = \text{milestone} \rightarrow (A_e < B_s \vee A_e = B_s) \right] \wedge \left[A_e = \neg \text{milestone} \rightarrow (A_e < C_s \vee A_e = C_s) \right] \wedge \\
& \left[B_e = \text{milestone} \rightarrow (B_e < B_s \vee B_e = B_s) \right] \wedge \left[B_e = \neg \text{milestone} \rightarrow (B_e < C_s \vee B_e = C_s) \right] \equiv \\
& (A_e = \text{milestone} \rightarrow A_e \leq B_s) \wedge (A_e = \neg \text{milestone} \rightarrow A_e \leq C_s) \wedge \\
& (B_e = \text{milestone} \rightarrow B_e \leq B_s) \wedge (B_e = \neg \text{milestone} \rightarrow B_e \leq C_s) \quad (A-15)
\end{aligned}$$

The above formula states that if at the time activity *A* is completed, *milestone* has been met, then the completion of activity *A* happens either some time before or at the same time activity *B* becomes enabled; but if at the time activity *A* is completed, *milestone* has been expired, then the completion of activity *A* happens either some time before or at the same time activity *C* becomes enabled. On the other hand, if at the time activity *B* is completed, *milestone* has been again met, activity *B* becomes enabled again; otherwise, activity *C* becomes enabled. Note that in the above formula, *milestone* is a zero-length interval at which a particular milestone is met.

A.3.2.2. Reo Milestone

Figure A-16 represents the Reo implementation of Milestone in Reo. We implement the Milestone pattern in Reo using the Deferred Choice connector. In Deferred Choice, according to some milestone, the decision is made whether activity *B* or *C* should be executed after activity *A*. If the milestone has not been met, the Deferred Choice will decide that activity *C* should be executed next. But if the milestone has been met, activity *B* will become enabled. Activity *B* will be executed continuously as long as that milestone has not expired. The milestone will be checked after each execution of *B*. When the milestone expires then activity *C* will become enabled.

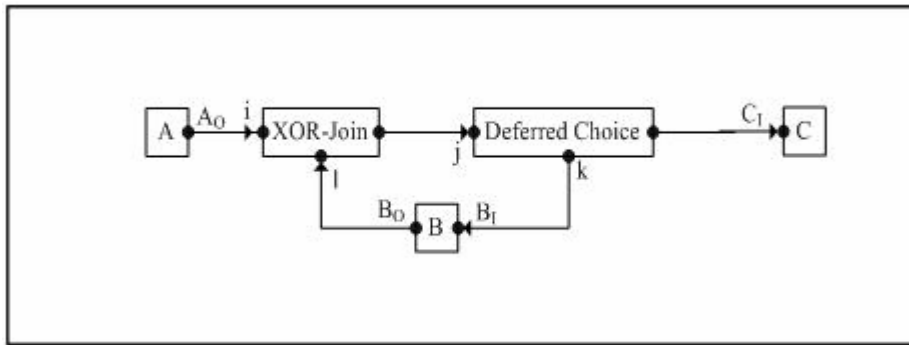


Figure A-5-16 Reo Milestone connector

Now we present the TDS behavior of the Milestone connector.

1. For the `sync` connector consisting of nodes A_o and i we have:

$$\langle d(A_o), \tau(A_o) \rangle \text{sync} \langle d(i), \tau(i) \rangle \equiv d(A_o) = d(i) \wedge \tau(A_o) = \tau(i)$$

2. For the `XOR-Join` connector consisting of nodes i , l , and j we have:

$$\begin{aligned}
& \text{XOR-Join}(\langle d(i), \tau(i) \rangle, \langle d(l), \tau(l) \rangle, \langle d(l), \tau(l) \rangle) \equiv \\
& \tau(i_0) \neq \tau(l_0) \wedge \\
& \left\{ \begin{array}{l} d(i_0) = d(j_0) \wedge \tau(i_0) \leq \tau(j_0) \wedge \\ \text{XOR-Join}(\langle d(i)', \tau(i)' \rangle, \langle d(l), \tau(l) \rangle, \langle d(j)', \tau(j)' \rangle) \quad \text{if } \tau(i_0) < \tau(l_0) \\ d(l_0) = d(j_0) \wedge \tau(l_0) \leq \tau(j_0) \wedge \\ \text{XOR-Join}(\langle d(i), \tau(i) \rangle, \langle d(l)', \tau(l)' \rangle, \langle d(j)', \tau(j)' \rangle) \quad \text{if } \tau(l_0) < \tau(i_0) \end{array} \right.
\end{aligned}$$

3. For the Deferred Choice connector consisting of nodes j , k and C_I we have:

$$\begin{aligned}
& \text{DeferredChoice}(\langle d(j), \tau(j) \rangle, \langle d(k), \tau(k) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv \\
& \left\{ \begin{array}{l} d(j) = d(k) \wedge \tau(j) \leq \tau(k) \quad \text{if } \textit{milestone} \\ d(j) = d(C_I) \wedge \tau(j) \leq \tau(C_I) \quad \text{if } \neg \textit{milesotne} \end{array} \right.
\end{aligned}$$

4. For the sync connector consisting of nodes k and B_I we have:

$$\langle d(k), \tau(k) \rangle \text{sync} \langle d(B_I), \tau(B_I) \rangle \equiv d(k) = d(B_I) \wedge \tau(k) = \tau(B_I)$$

5. For the sync connector consisting of nodes B_O and l we have:

$$\langle d(B_O), \tau(B_O) \rangle \text{sync} \langle d(l), \tau(l) \rangle \equiv d(B_O) = d(l) \wedge \tau(B_O) = \tau(l)$$

By composing five formulas above, we obtain the TDS formula for Reo Milestone:

$$\text{Milestone}(\langle d(A_O), \tau(A_O) \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(C_I), \tau(C_I) \rangle) \equiv (A-16)$$

$$\tau(A_{O0}) \neq \tau(B_{O0}) \wedge$$

$$\left\{ \begin{array}{l} d(A_{O0}) = d(j_0) \wedge \tau(A_{O0}) \leq \tau(j_0) \wedge \\ \left\{ \begin{array}{l} d(j_0) = d(B_{I0}) \wedge \tau(j_0) \leq \tau(B_{I0}) \quad \text{if } \text{milestone} \\ d(j_0) = d(C_{I0}) \wedge \tau(j_0) \leq \tau(C_{I0}) \quad \text{if } \neg \text{milestone} \end{array} \right. \wedge \quad \text{if } \tau(A_{O0}) < \tau(B_{O0}) \\ \text{Milestone}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(B_I)', \tau(B_I)' \rangle, \langle d(B_O)', \tau(B_O)' \rangle, \langle d(C_I), \tau(C_I) \rangle) \end{array} \right.$$

$$\left\{ \begin{array}{l} d(B_{O0}) = d(j_0) \wedge \tau(B_{O0}) \leq \tau(j_0) \wedge \\ \left\{ \begin{array}{l} d(j_0) = d(B_{I0}) \wedge \tau(j_0) \leq \tau(B_{I0}) \quad \text{if } \text{milestone} \\ d(j_0) = d(C_{I0}) \wedge \tau(j_0) \leq \tau(C_{I0}) \quad \text{if } \neg \text{milestone} \end{array} \right. \wedge \quad \text{if } \tau(B_{O0}) < \tau(A_{O0}) \\ \text{Milestone}(\langle d(A_O)', \tau(A_O)' \rangle, \langle d(B_I), \tau(B_I) \rangle, \langle d(B_O), \tau(B_O) \rangle, \langle d(C_I)', \tau(C_I)' \rangle) \end{array} \right.$$

A.4. Cancellation Patterns

A.4.1. Cancel Activity

This pattern represents a situation in a workflow when an enabled activity is disabled by another activity or an external event. This means that the thread which is waiting for the execution of that activity is removed. It is assumed that the cancellation of an activity refers to a single instance of that activity. As an example, normally, a design is checked by two groups of engineers. However, to meet deadlines it is possible that one of these checks is withdrawn to be able to meet a deadline.

A.4.1.1. Workflow Cancel Activity

Figure A-17 represents workflow Cancel Activity pattern in which after completion of activity A, based on a signal from environment, activity C will be withdrawn, although it has been enabled.

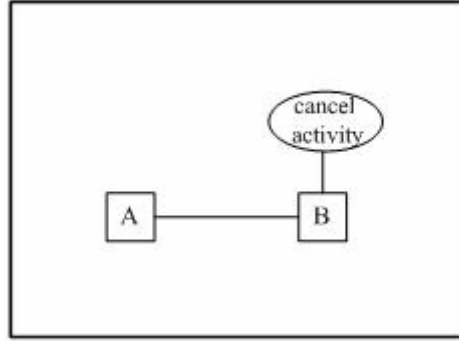


Figure A-5-17 workflow Cancel Activity

Formula (A -17) represents the PITL formula for this pattern. *cancel* is the time point at which *cancel* signal is sent from environment. *withdraw(C)* is the time point at which activity *C* is withdrawn.

$$\begin{aligned}
 (\text{cancel During } C) \rightarrow \text{withdraw}(C) \text{ Equal } \text{cancel} & \equiv & (A-17) \\
 C_s < \text{cancel} < C_e \rightarrow \text{withdraw}(C) = \text{cancel} &
 \end{aligned}$$

The above formula states that if the *cancel* is received during the execution of activity *C*, then activity *C* is withdrawn and the time point it is withdrawn is equal to the point of time the *cancel* signal has been received.

A.4.1.2. Reo Cancel Activity

Figure A-18 represents the Reo implementation of Cancel Activity in Reo. Since Reo is not aware of what happens inside component instances and does not have any control over them, when an activity becomes enabled, we cannot stop or cancel it directly. So we have to implement this pattern indirectly by using a shadow activity and the Deferred Choice pattern. Both the main activity and the shadow activity are preceded by a Deferred Choice. Assume that in a workflow case, activity *C* should be executed after activity *A* and there is the possibility that according to some cancel signal from the environment, the execution of activity *C* be canceled. We introduce the shadow activity *B* for this example. So, whenever activity *C* should be canceled, in the Deferred Choice

connector, activity *B* becomes enabled instead of *C* (the execution of activity *B* is not a real execution since it is a shadow activity).

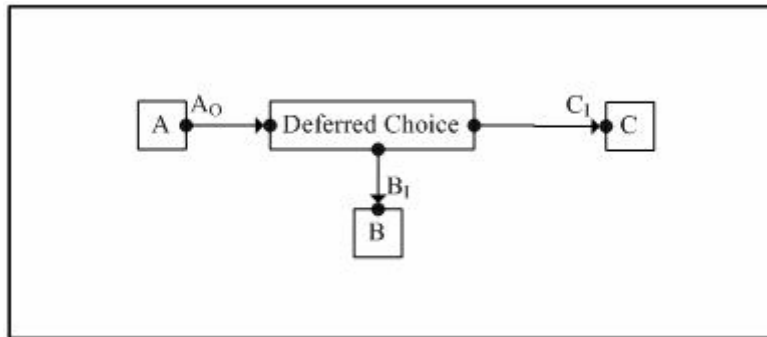


Figure A-5-18 Reo Cancel Activity connector

The TDS formula for Reo Cancel Activity is:

$$\text{CancelActivity}(\langle d(A_o), \tau(A_o) \rangle, \langle d(B_i), \tau(B_i) \rangle, \langle d(C_i), \tau(C_i) \rangle) \equiv \quad (A-18)$$

$$\begin{cases} (d(A_o) = d(B_i) \wedge \tau(A_o) \leq \tau(B_i)) & \text{if } \textit{cancel} \\ (d(A_o) = d(C_i) \wedge \tau(A_o) \leq \tau(C_i)) & \text{if } \neg \textit{cancel} \end{cases}$$

The above formula looks the same as Deferred Choice pattern: here the milestone is the cancellation signal from environment and activity *B* is not a real activity; it is just a shadow activity to cancel the execution of activity *C*.

A.4.2. Cancel Case

This pattern represents a situation in a workflow when the whole workflow instance is cancelled by an activity or an external event. For the cancellation of an entire case, all instances of each activity are cancelled. As an example, in an insurance processing system, a customer withdraws an insurance claim before the final decision is made.

A.4.1.1. Workflow Cancel Case

Figure A-19 represents workflow Cancel Case pattern in which, based on a signal from environment, the whole workflow case should be cancelled.

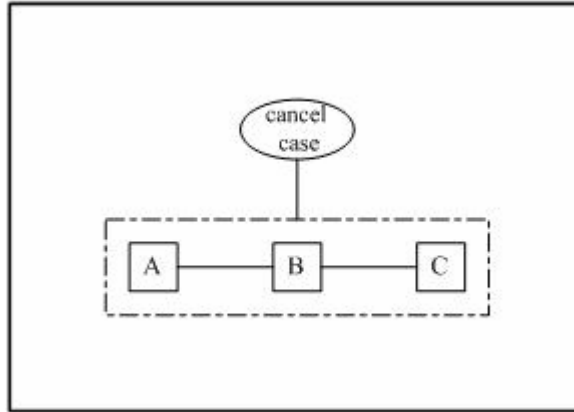


Figure A-5-19 Workflow Cancel Case

Formula (A -19) represents the PITL formula for this pattern. *cancel* is the time point at which *cancel* signal is sent from environment. *withdraw(w)* is the time point at which workflow case *w* is withdrawn.

$$\begin{aligned}
 (\textit{cancel} \textbf{During} w) \rightarrow \textit{withdraw}(w) \textbf{Equal} \textit{cancel} &\equiv & (A-19) \\
 w_s < \textit{cancel} < w_e \rightarrow \textit{withdraw}(w) = \textit{cancel} &
 \end{aligned}$$

The above formula states that if the *cancel* is received during the execution of workflow instance *w*, then *w* is withdrawn and the time point it is withdrawn is equal to the point of time the *cancel* signal has been received.

A.4.1.2. Reo Cancel Case

We implement the Cancel Case pattern using the Cancel Activity pattern. Each activity in the workflow case that may have the possibility to be canceled is preceded by a cancel activity pattern. When the whole case should be canceled, a message will be send to all activities and all shadow activities in the workflow case will become enabled instead of the real activities.

Appendix B: Overview of Point Interval Temporal

Logic

Point interval temporal logic (PITL) [40][42] is an extension of Allen’s interval logic [19][20] that involves both descriptions of points and intervals of time. In interval logic introduced by Allen the primitive notion of time is an *interval* which is a non-zero length of time [20]. In PITL, intervals with zero lengths, i.e. *points*, are also introduced.

B.1. Basic Concepts

Now we represent the basic definitions of PITL [40][42].

Interval: An interval is denoted by a symbol, which can be a letter or a digit, and is defined as $X = [sx, ex]$, where sx and ex are positive integers and $ex \geq sx$. In this definition sx denotes “start of x ” and ex denotes “end of x ”. The two bounds are abstract notions and may not have numerical values assigned to them. The intervals considered are closed intervals.

Interval Length: The length of an interval $X = [sx, ex]$, denoted by $|X|$, is defined as $|X| = ex - sx$.

Point: An interval X with $|X| = 0$ is a point interval; or point, denoted as $[px]$. In a system description, a point may be used to signify an occurrence or an event.

Temporal Relation R_i : The temporal relation R_i is a truth functional binary relation defined as: $R_i : I \times I \rightarrow \{T, F\}$ where I is the set of intervals and T and F are the Boolean values, *true* and *false*.

Set of Temporal Relations R : R represents the set of temporal relations R_i and is given as $R = \{\text{Before, Meets, Overlaps, Starts, During, Finishes, Equals}\}$.

The definition of temporal relations for both zero length and non-zero length intervals is given in figure B-1.

I. $X = [sx, ex], Y = [sy, ey]$		
X Before Y	\equiv	$ex < sy$
X Meets Y	\equiv	$ex = sy$
X Overlaps Y	\equiv	$sx < sy, sy < ex, ex < ey$
X Starts Y	\equiv	$sx = sy, ex < ey$
X During Y	\equiv	$sx > sy, ex < ey$
X Finishes Y	\equiv	$sy < sx, ex = ey$
X Equals Y	\equiv	$sx = sy, ex = ey$
II. $X = [px], Y = [py]$		
X Before Y	\equiv	$px < py$
X Equals Y	\equiv	$px = py$
III. $X = [px], Y = [sy, ey]$		
X Before Y	\equiv	$px < sy$
Y Before X	\equiv	$sx = sy, ex = ey$
X Starts Y	\equiv	$px = sy$
X During Y	\equiv	$sx < px < ey$
X Finishes Y	\equiv	$px = ey$

Figure B-1 Point Interval Temporal Logics Operators

We may have combinations of zero length or non-zero length intervals related to each other by the seven operations above. Figure B-1 shows the possible relationships. The first case which is shown in part one of the figure B-1 is that both intervals are non-zero length; all relationships are applicable to these intervals with the given definition.

The second case is that both intervals are zero length; they are points of time. Only **Before** and **Equals** operations are applicable; other relations are obviously nonsense in case of points of time. The third case is when one is a point of time and the other is a non-zero length interval; the possible relations are **Before**, **Starts**, **During**, and **Finishes**.

References

- [1] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. Bulletin of the EATCS, 62:222--259, 1997. Available at URL: www.cwi.nl/#janr.
- [2] B. Kiepuszewski, A.H.M. ter Hofstede, W.M.P. van der Aalst. Fundamentals of Control Flow in Workflows. In *QUT Technical report, FIT-TR-2002-03*. Queensland University of Technology, Brisbane. 2002.
- [3] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases*, 3(2):119– 153, 1995.
- [4] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):96.107, February 1992.
- [5] David Hollingsworth. Workflow Management Coalition, The Workflow Reference Model. Document Number TC00-1003. Document Status - Issue 1.1. 19-Jan-95.
- [6] Dong Yang, Shen-sheng Zhang: Modeling Workflow Process Models with Statechart. *ECBS 2003*: 55-61.
- [7] E. A. Emerson. Temporal and modal logics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume Volume B, Formal Methods and Semantics, pages 995-1072. Elsevier Science Publishers and The MIT Press, 1990.
- [8] Eshuis, R. and R. Wieringa, Verification support for workflow design with UML activity graphs, in: Proc. 24th Intl. Conf. on Software Engineering (ICSE) (2002), pp. 166–176.
- [9] Eyal Oren, Armin Haller. Formal Frameworks for Workflow Modeling. Technical Report. Digital Enterprise Research Institute (DERI). 2005.
- [10] Farhad Arbab, Christel Baier, Jan J. M. M. Rutten, Marjan Sirjani. Modeling Component Connectors in Reo by Constraint Automata: (Extended Abstract). *Electr. Notes Theor. Comput. Sci.* 97: 25-46, 2004.
- [11] Farhad Arbab, Jan J. M. M. Rutten: A Coinductive Calculus of Component Connectors. *WADT 2002*: 34-55.

- [12] Farhad Arbab. A Behavioral Model for Composition of Software Components.
- [13] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14(3): 329-366, 2004.
- [14] G. A. Papadopoulos and F. Arbab, 'Coordination Models and Languages', *Advances in Computers*, Academic Press, Vol. 46: The Engineering of Large Systems, pp. 329-400. September 1998.
- [15] G. Alonso, D. Agrawal, A. Abbadi, C. Mohan: Functionality and Limitations of Current Workflow Management Systems, *IEEE Expert* Vol. 12, No. 5, 1997.
- [16] H. Davulcu, M. Kifer, C. Ramakrishnan, and I. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems*, pp. 25–33. 1998.
- [17] J. Wainer, Logic representation of processes in work activity coordination, *Proceedings of the ACM Symposium on Applied Computing, Coordination Track*. 203-209. 2000.
- [18] James F Allen. Time and Time Again: The Many Ways to Represent Time. *International Journal of Intelligent Systems*, 6(4): pp341-355, July 1991.
- [19] James F. Allen and George Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531-579, October 1994.
- [20] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832-843, 1983.
- [21] Li, H., Yang, Y., Chen, T.Y.: Resource Constraints Analysis of Workflow Specifications. *The Journal of Systems and Software*, Elsevier, in press
- [22] M. Dumas and A. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *Proc. of the International Conference on the Unified Modeling Language (UML)*. Toronto, Canada, October 2001. Springer Verlag.
- [23] M. Fowler, K. Scott, *UML distilled*, 2nd Edition, AddisonWesley, Harlow, 2000.
- [24] Mashhood Ishaque, Abbas K. Zaidi. TIME-SENSITIVE PLANNING USING POINT-INTERVAL LOGIC. Student Paper, 10th International Command and Control Research and Technology Symposium, The Future of C2, McLean, VA. June 2005.

- [25] N. R. Adam, V. Atluri, W. Huang. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems (JIIS), Special Issue on Workflow and Process Management*, 10(2):131–158, March/April 1998.
- [26] Nikolay K. Diakov, Farhad Arbab. Compositional Construction of Web Services Using Reo. *WSMAI 2004*: 49-58.
- [27] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and N. Russell. Pattern-based Analysis of UML Activity Diagrams. (PDF, 149 Kb). BETA Working Paper Series, WP 129, Eindhoven University of Technology, Eindhoven, 2004.
- [28] R. Eshuis and J. Dehnert. Reactive petri nets for workflow modeling. In W. M. P. van der Aalst and E. Best, (eds.) *Application and Theory of Petri Nets 2003*, vol. 2679 of *Lecture Notes in Computer Science*, pp. 295–314. Springer-Verlag, Berlin, Berlin, 2003.
- [29] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. Int. Thomson Press, 1996.
- [30] S. Mukherjee, et al. Logic-based approaches to workflow modeling and verification. In J. Chomicki, R. van der Meyden, and G. Saake, (eds.) *Logics for Emerging Applications of Databases*, chap. 5, pp. 167–202. Springer-Verlag, Berlin, 2004.
- [31] T. Murata, "Petri nets - properties, analysis, and applications"; *Proc. IEEE*, vol.77, no.4, pp.541-580, 1989.
- [32] W. M. P. van der Aalst and A. H. M. ter Hofstede. Workflow patterns: On the expressive power of (Petri-net-based) workflow languages. In K. Jensen, (ed.) *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*. 2002.
- [33] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
- [34] W. M. P. van der Aalst. Making work flow: On the application of petri nets to business process management. In J. Esparza and C. Lakos, (eds.) *23rd International Conference on Applications and Theory of Petri Nets*, vol. 2360 of *Lecture Notes in Computer Science*, pp. 1–22. Springer-Verlag, Berlin, 2002.
- [35] W. M. P. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

- [36] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
- [37] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. *Workow Patterns*. QUT Technical report, FIT-TR-2002-02 (to appear in *Distributed and Parallel Databases*), Queensland University of Technology, Brisbane, 2002. <http://www.tm.tue.nl/it/research/patterns>.
- [38] W.M.P. van der Aalst, M. Weske, G. Wirtz. *Advanced Topics in Workflow Management: Issues, Requirements, and Solutions*. *Journal of Integrated Design and Process Science*. Volume 7, Number 3. Austin: Society for Design and Process Science 2003.
- [39] Workflow Management Coalition, *Terminology and Golossary*, WfMC (1999). Document Number WFMC-TC-1011, Document Status - Issue 3.0.
- [40] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, NEW York, pp.1816, 258-261, 1992.
- [41] Zaidi, A. K. 2002. A Temporal Programmer for Time-Sensitive Modeling of Discrete-Event Systems. In *Proceedings of IEEE - Systems, Man, and Cybernetics Society 2000 Meeting*. Nashville, TN.
- [42] Zaidi, A., 1999. On temporal logic programming using petri nets. *IEEE Transactions on Systems, Man, and Cybernetic s, Part A: Systems and Humans* 29 (3), 245–254.