



Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

Dissertação de Mestrado em Engenharia Informática

1º Semestre, 2009/2010

Web Services Recovery Mechanisms

Nº 28022 Rui Filipe Vital Guerreiro da Costa

Orientadora

Prof. Carla Ferreira

22 de Fevereiro de 2010

Nº do aluno: 28022

Nome: Rui Filipe Vital Guerreiro da Costa

Título da dissertação:

Mecanismos de Recuperação para Serviços Web

(Web Services Recovery Mechanisms)

Palavras-Chave:

Serviços Web

Mecanismos de Recuperação

Transacções de negócio

WS-BPEL

Keywords:

Web Services

Recovery Mechanisms

Business transactions

WS-BPEL

## **Agradecimentos**

Agradeço a todos os meus amigos e familiares aos quais não pude dedicar mais tempo durante a criação desta dissertação. Tenho que agradecer especialmente à minha namorada Sandra, ao meu irmão Pedro e aos meu sócio Paulo pelo apoio que me deram.

Agradeço também à minha professora Carla Ferreira por me orientar e pela paciência que mostrou.

Dedico esta tese aos meus pais Jonas e Otília que já não estão entre nós.

## **Resumo**

---

No contexto dos serviços Web não é possível utilizar o conceito usual de transacções que respeitam as propriedades ACID devido a vários factores. Por exemplo, o facto de as transacções de negócio terem em geral uma duração elevada que pode variar entre vários dias a vários meses, ou então, por envolverem a coordenação e interacção de actividades executadas por diferentes parceiros. Tendo estes factores em consideração, a propriedade de atomicidade não é preservada e consequentemente os mecanismos usuais de recuperação (tal como rollback) não podem ser usados.

Para transacções de negócio, o tratamento de falhas pode ser feito através de mecanismos de compensação. Estes mecanismos definem acções que compensam outras acções que não podem ser revertidas automaticamente. Esta dissertação tem como objectivo definir um conjunto de padrões que representam a utilização comum dos mecanismos de recuperação ao nível das transacções de negócio. Para mostrar como funcionam os mecanismos de recuperação vai ser definida uma notação gráfica de fácil compreensão de modo a estar acessível a pessoas com diferentes níveis de formação.

---

## **Abstract**

---

In web services context it is not possible to use the usual concept of ACID transactions because of several factors. For instance, business transaction in general have a long duration that can be extended to several months or can involve the coordination and interaction of activities executed by different partners. In these cases, atomicity is not preserved, therefore the usual recovery mechanisms cannot be used, like the rollback.

In business transaction, failure treatment can be made by compensation mechanisms in which are defined actions of compensation for the actions that cannot be reverted automatically. The goal of this dissertation is to define a set of patterns that represent the common use of the recovery mechanisms at business level. A graphical notation of easy comprehension will be developed to show how the recovery mechanisms work to all kind of people with different background formation.

---

## Acronym

Acronym	Meaning
ACID	Atomicity, Consistency, Isolation and Durability
BPC	Business Process Choreographer
FSP	Finite State Process
LRT	Long Running Transactions
LTS	Labeled Transition System
OASIS	Organization for the Advancement of Structured Information Standards
oWFN	Open Workflow Nets
WF-net	Workflow Net
WPN	Workflow Petri Net
WoPeD	Workflow Petri Net Designer
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Service Definition Language
WSFL	Web Service Flow Language

## Index

---

<b>1. Introduction .....</b>	<b>14</b>
1.1 Motivation .....	14
1.2 Expected Contribution.....	15
1.3 Document Organization .....	15
<b>2. Exception Mechanisms .....</b>	<b>16</b>
2.1 Transactions .....	16
2.2 Long Running Transactions (LRT).....	17
2.3 Failure Handling and Cancelation Mechanisms.....	17
2.3.1 Exception Handling Mechanism .....	17
2.3.2 Concepts from Advanced Transactional Models .....	18
2.4 Summary .....	18
<b>3. WS-BPEL .....</b>	<b>19</b>
3.1 Business Processes .....	19
3.2 Language Constructs .....	20
3.2.1 Basic Activities .....	20
3.2.2 Structured Activities.....	22
3.2.3 Scopes .....	23
3.3 Recovery Mechanisms .....	24
3.4 Analysis of Compensation Mechanisms .....	24

3.5	Other Business Specifications .....	25
3.5.1	BizTalk .....	26
3.5.2	WebSphere .....	26
3.6	Summary .....	27
<b>4.</b>	<b>WS-BPEL Formalization.....</b>	<b>28</b>
4.1	Formal Languages with Graphical notation .....	28
4.1.1	Petri Nets .....	28
4.1.2	Statecharts .....	29
4.1.3	Finite State Process .....	30
4.2	Related Work .....	31
4.2.1	Petri Nets .....	31
4.2.2	Finite State Process .....	34
4.2.3	Workflow Nets .....	36
4.3	Summary .....	40
<b>5.</b>	<b>Mapping WS-BPEL to Workflow Petri Nets.....</b>	<b>41</b>
5.1	WPN Components.....	41
5.2	Basic Activities .....	43
5.3	Structured Activities.....	49
5.4	Scopes .....	56
5.4.1	Simple .....	56
5.4.2	Fault Handlers .....	56
5.4.3	Compensation Handlers .....	59
5.5	Summary .....	60
<b>6.</b>	<b>Booking Agency Case Study.....</b>	<b>61</b>
6.1	WS-BPEL.....	63
6.1.1	Availability Activities .....	64
6.1.2	Booking Activities.....	65



6.1.3	Transfer Activities .....	67
6.1.4	Cancellation Activities .....	70
6.2	Workflow Petri Nets.....	70
6.2.1	Direct Mapping .....	70
6.2.2	Overview Mapping.....	78
6.3	Other Compensation Features .....	82
6.4	Comparing BPEL2oWFN .....	84
6.5	Summary .....	86
<b>7.</b>	<b>Conclusion.....</b>	<b>87</b>
7.1	Contribution and work limitations .....	88
7.2	Future Work .....	88
	<b>Bibliography.....</b>	<b>89</b>

## Figure Index

---

Figure 3.1 - Graphical representation of nested scopes .....	23
Figure 4.1 - Illustration of a Petri Net firing rule .....	29
Figure 4.2 - Statechart example .....	30
Figure 4.3 – Terminate Activity in Petri Net.....	33
Figure 4.4 – Pattern “receiving a message” .....	34
Figure 4.5 – Sequence example .....	35
Figure 4.6 - Workflow condition blocks .....	36
Figure 4.7 - Workflow triggers notation.....	37
Figure 4.8 - WoPeD screenshot of property editor.....	38
Figure 4.9 - WoPeD screenshot of token game .....	38
Figure 4.10 - WoPeD screenshot of soundness analysis .....	39
Figure 5.1 - WPN pattern for receive activity .....	43
Figure 5.2 - WPN pattern for reply activity.....	44
Figure 5.3 - WPN pattern for invoke activity .....	45
Figure 5.4 - WPN pattern for assign activity.....	45

Figure 5.5 - WPN pattern for throw activity .....	46
Figure 5.6 - WPN pattern for wait activity .....	47
Figure 5.7 - WPN pattern for empty activity .....	47
Figure 5.8 - WPN pattern for exit activity .....	48
Figure 5.9 - WPN pattern for Rethrow activity .....	48
Figure 5.10 - WPN pattern for the compensate activity .....	49
Figure 5.11 - WPN pattern for sequence activity .....	49
Figure 5.12- WPN pattern for if activity .....	50
Figure 5.13 - WPN pattern for while activity .....	51
Figure 5.14- WPN pattern for repeat until activity .....	52
Figure 5.15 - WPN pattern for pick activity .....	53
Figure 5.16 - WPN pattern for flow activity .....	54
Figure 5.17 - WPN pattern for foreach activity .....	55
Figure 5.18 - WPN pattern for a scope .....	56
Figure 5.19 - WPN pattern for a scope with fault handlers .....	57
Figure 5.20 - WPN pattern for the activity within the failure handlers .....	58
Figure 5.21 - WPN pattern for a scope with compensation and fault handlers .....	59
Figure 6.1 - Activity diagram for the booking process .....	61
Figure 6.2 - Activity diagram to cancel a booking .....	62
Figure 6.3 - Graphical overview of the booking agency process .....	63
Figure 6.4 - Graphical representation of the AvailabilityScope .....	64

Figure 6.5 - Graphical representation of the ScopeBooking .....	66
Figure 6.6 - Graphical representation of the ScopeBooking compensation .....	67
Figure 6.7 - Graphical representation of the ScopeTransfer .....	68
Figure 6.8 - Graphical representation of the Failure and Compensation handlers of ScopeTransfer.....	69
Figure 6.9 - Graphical representation of the ScopeCancelation and failure handlers .....	70
Figure 6.10 - WPN mapping for the ScopeAvailability .....	71
Figure 6.11 - WPN mapping for the ScopeBooking .....	72
Figure 6.12 - WPN mapping for the compensation handlers of ScopeBooking .....	73
Figure 6.13 - WPN mapping for the ScopeTransfer.....	75
Figure 6.14 - WPN mapping for the failure handler for the ScopeTransfer.....	76
Figure 6.15 - WPN mapping for the compensation handler for the ScopeTransfer.....	76
Figure 6.16 - WPN mapping for the ScopeCancelation .....	77
Figure 6.17 - WPN mapping for the failure handler of the ScopeCancelation .....	77
Figure 6.18 - Overview of the WPN mapping for the compensation of the booking agency process .....	78
Figure 6.19 - Detailed WPN mapping for the booking agency process .....	80
Figure 6.20 - Compensation within Handlers examples.....	83
Figure 6.21 - Graphical representation of ScopeCancelation using BPEL2oWFN and Graphviz .....	84
Figure 6.22 - Graphical representation of ScopeCancelation using this work mapping .....	85

## Table Index

---

Table 1 - Basic Activities of WS-BPEL.....	21
Table 2 - Structured Activities of WS-BPEL .....	22
Table 3 - Some FSP operators .....	31
Table 4 - WPN components .....	42

## **1. Introduction**

The internet has revolutionized the way we interact with the world around us. Now we can stay at home and do things that once needed our presence to be achieved. We can buy groceries, manage bank accounts or book vacations among lots of others operations. These operations usually, in order to be accomplished, need to exchange data and messages between different systems used by the parties involved. To achieve that goal, web services provide the interoperability between companies and their systems. To do so, some rules must be taken in consideration by all participants, rules that are defined in the business process. WS-BPEL[1] has become a standard to define those rules, and is supported by the main corporations that develop and provide means to create web services.

Web services need to have recovery mechanisms, compensation methods and failure treatment to provide an efficient and secure usage by all users. Usually web services depend on database support to store data, therefore some of these concepts where inherited from database design.

The recovery mechanisms created for web services become more complex when a process is distributed between different partners. An error on one web service will influence the others, and if not handled correctly, it may lead to unforeseen events that may damage the participants relationships.

### **1.1 Motivation**

This work was proposed because of the lack of precise information regarding one of the most import aspects in the development of web services, the recovery mechanisms. Although WS-BPEL is a standard, it is still under development. So in order to suppress some needs that might not be covered by the standard, the existing web services tools need to adapt business

process recovery mechanisms. Those changes in business process implementations have not been studied in detail, therefore it has become part of this work.

## **1.2 Expected Contribution**

This work is expected to provide a mapping in a formal language of the business process recovery mechanisms. Doing so, it will improve the knowledge of business process in web services. It should provide a deep insight of the recovery mechanisms associated with web services, their definitions, flaws and limitations. The mapping should also provide a way to show even non experts, the steps that the business process goes thru to execute the tasks that is made for. This includes besides the recovery behavior, also the normal behavior.

## **1.3 Document Organization**

This document is organized in the following manner: the second chapter introduces some concepts used in exception mechanisms that are the basis of recovery mechanisms in web services. The third chapter makes an overview of the business process specifications including the different existing activities and some tools that can model and implement business process. The fourth chapter contains the related work that shows a few graphical formal languages that could be used to give a formal notation to the findings of this work. Fifth chapter provides a mapping for the different activities present in the business process into the formal language chosen. The sixth chapter introduces a case study do demonstrate the recovery mechanisms in the business process. The last chapter provides a conclusion for this thesis.

## 2. Exception Mechanisms

There are many approaches to control exceptions and cancelations. This chapter presents some of those concepts on which web services depend. Concepts like transactions that are the basis of the connections between web services and some common failures handling and cancellations mechanisms.

### 2.1 Transactions

Transactions are usually used in database systems with concurrent access by different clients. The goal of using transactions is to group a collection of operations such that once executed, all operations succeed or none does. To ensure that the data accessed and modified by the clients is always consistent, the database systems must maintain a few properties regarding the transactions. These properties are known by ACID [2] which stands for:

- **Atomicity:** All operations within the transaction must execute successfully as if it were only one. If not, then the changes made by the transaction must be undone, and in this case it would appear like the transaction never occurred. The atomicity is usual guaranteed by a locking system;
- **Consistency:** The consistency of the database must be preserved by the transaction.
- **Isolation:** Even if multiple transactions are executed concurrently, each transaction must be unaware that the others are running. Intermediate affects of a transaction must be invisible to the others;
- **Durability:** This property assures that if a transaction is successful, then all the changes made by that transaction will persist even if occur a system failure.



## 2.2 Long Running Transactions (LRT)

Most transactions are non-interactive and of short duration [3]. Whenever a human interacts with a transaction, it becomes a long-running transaction because the human response time is slower than computer speed. In such cases, the transaction may last hours, days or months just because it needs human intervention. From this type of transactions some problems surface: the ability to abort subtasks, exposure to uncommitted data, recoverability and performance.

- **Subtasks:** The user may wish to abort a subtask, but not the entire transaction.
- **Exposure of uncommitted data:** The data generated and displayed to a user in a long-duration transaction are uncommitted, so concurrent transactions may be forced to read uncommitted data.
- **Recoverability:** This type of transaction cannot abort because of system crash. It must be recovered to a consistent state that existed prior to the crash, without affecting human work.
- **Performance:** The most costly resource is the user. So in order to optimize the user interaction within the transaction, the tasks that take longer to execute should be predictable, so that users can manage their time.

Because of these features, the common failures handling and cancellations mechanisms must be adapted for these transactions.

## 2.3 Failure Handling and Cancellation Mechanisms

Like other processes, the business process must provide a way to handle exceptions and cancel the execution of some work.

### 2.3.1 Exception Handling Mechanism

When a condition occurs that changes the normal execution of code, then an exception has occurred. If it is not handled, it usually aborts the execution of a program. To prevent this, in

most common programming languages [4] exists built-in support for exception and exception handling. Usually the programmer can define where the exceptions are caught and which exceptions must be treated. It can be applied to a single operation or to multiple operations. If an exception is caught, then the program must execute the appropriate code to handle it, or at least warn the user of the problem.

### **2.3.2 Concepts from Advanced Transactional Models**

The term Saga [5] is applied in context of relational database and it is used to refer a long-running transaction that can be divided into a collection of sub-transactions. Those sub-transactions can be interleaved in any way with other transactions. Each sub-transaction in a saga has the ACID properties and should have a compensation transaction which is called when a failure occurs. Unlike the rollback in database, this compensation transaction may not return the system to the initial state. Sagas may be seen as nested transactions [6] but with two major differences:

- a) Only permits two levels of nesting: top level saga and simple transactions;
- b) Sagas may view the partial results of other sagas (full atomicity is not provided).

Sagas provide two types of compensation: the *backward recovery* and the *forward recovery*. If one of the nested transactions fails, when using *backward recovery*, the compensation transactions of the previous successful nested transaction will be called in the reverse order of execution. When using *forward recovery*, beside the compensation transactions, it needs save-points defined within the saga. It does the same as *backward recovery*, but it stops the recovery at the save point, and tries to run the transactions again from that point forward.

## **2.4 Summary**

This chapter showed some concepts used in many existing systems, including business process and web services. Introduced the notion of short and long running transactions. The exceptions in programming languages and the definition of Sagas. The next chapter will give an overview of business process which is the base of this work.

### 3. WS-BPEL

It is a language for specifying business process on web services and it stands for *Web Services Business Process Execution Language*. It is based on *XLang* [7] by Microsoft and *Web Services Flow Language* (WSFL) [8] by IBM and uses XML [9] syntax. It has become a language standard for the Orchestration of logic execution in Web services applications, supported by Microsoft, IBM, BEA Systems, SAP, among others. *Organization for the Advancement of Structured Information Standards* (OASIS) manages the standardization process, and WS-BPEL is currently in the 2.0 version. Originally it was named BPEL4WS and because of that, both names are used to refer this business process language.

WS-BPEL can describe an abstract business process which serves as a descriptive role and therefore is not intended to be executed. It can also describe executable business processes that model the actual behavior of the participants. That behavior consists of the interactions between the process and its partners. Every partner uses a Web Service interface to interact and each interaction must be coordinated to achieve a business goal. This language also provides ways for dealing with exceptions and failures.

#### 3.1 Business Processes

The business process coordinates the interaction with other web services. This coordination is contained in a WS-BPEL file that contains the way our web service communicates with other web services. But before defining the coordination, first it must be declared the message types, operation names and locations of the different partners involved. This is done using a WSDL [10] files. WSDL is another XML formatted file that contains the information provided by a partner in order to communicate with other web services. Having these files, the definition of the business process can start. The WS-BPEL file is usually composed by the partner links, variables and the process definition. It can also have fault handlers defined for


the process. The partner links contains the parties involved in the business process and their roles in the relationship. The variables contains the state of the business process. The state can include, for instance, the messages received and sent to partners. The variables can be of several types and contain simple or complex data. The variable names must be unique and the declaration can be global or be part of a scope. Some of the activities of WS-BPEL must have associated variables in order to function. The process definition contains the description for the normal behavior of the business process. Fault handlers define the activities that must be performed when something goes wrong.









## 3.2 Language Constructs

WS-BPEL language is composed by many different XML tags. This section will focus part of the language that is relevant to this thesis. Defining *Variables*, *Correlations*, *Links* or *Termination handlers* are not covered.

### 3.2.1 Basic Activities

There are many different activities in WS-BPEL. The basic activities describe the basic steps of the process behavior and are detailed in Table 1. This includes also associated symbols existing in Oracle JDeveloper [11] that will appear latter on in the Case Study chapter.

Activity	Symbol	Goal
Invoke		This activity invokes an operation offered by a partner. It can be a one-way or a request-response operation. This activity sends a message to the partner that must be appropriate to the operation invoked. If the operation is request-response, the invoke will wait for the response message.

<b>Receive</b>		This is an asynchronous activity that waits for a partner to invoke an specific operation. This is done by receiving a message sent by a partner.
<b>Reply</b>		The reply activity is used in conjunction with the receive activity when a request-response is invoked by a partner. This will send a message to the partner with the appropriate message.
<b>Assign</b>		This activity passes data from one variable to another.
<b>Throw</b>		This activity is used to signal an explicit fault to the business process. The fault thrown must be thread by a fault handler.
<b>Wait</b>		This activity delays the execution of the process. This can be done by waiting for a period of time or until a deadline.
<b>Empty</b>		This activity does nothing.
<b>Exit</b>		This activity is used to end a business process without handling faults, termination or compensations.
<b>Re-throw</b>		This activity is used in fault handlers to re-throw a previous fault.
<b>Extension Activity</b>		This activity defines new activities that are not defined by the WS-BPEL specification. It will not be covered.
<b>Compensate</b>		This activity calls the compensation of a previously executed scope. This cannot be called within the normal execution.

**Table 1 - Basic Activities of WS-BPEL**

### 3.2.2 Structured Activities

Structured activities describe a way a collection of activities are executed. This activities can be a composition of basic and other structured activities. These activities are detailed in Table 2. Like the previous table, it also has symbols used in the case study chapter.






Activity	Symbol	Goal
Sequence		This activity contains one or more activities that are executed sequentially. This activity ends when all activities ends.
IF		The if activity consists of a list of conditional branches. If one of the conditions is true the associated activity is executed.
While		The while activity repeats an activity while a condition is true. This condition is evaluated at the beginning of each interaction.
Repeat Until		This activity performs almost the same operation of the while activity. The difference is that the conditions is evaluated at the end of each interaction and is executed at least one time.
Pick		The Pick activity waits for one event from a set of events. If the event occurs, the associated activity will be executed and all other events will no longer be accepted.
Flow		The flow activity provides a way to run concurrent activities and synchronization.
For Each		The ForEach activity executes a scope activity N + 1 times.

Table 2 - Structured Activities of WS-BPEL

### 3.2.3 Scopes

A scope is a collection of activities that are logically put together and can have local variables, fault handling, compensation handling among other constructs. The constructs that can be used by a scope are nested hierarchically and follow a few rules. A scope requires a primary activity that defines its normal behavior. This primary activity shares the context of the scope and usually is a structured activity that can have many levels. Scopes can also be nested with other scopes. In Figure 3.1 is shown a graphical representation of a possible organization of a WS-BPEL process. In this example, the process has a *Scope 1* that has two nested scopes, *Scope 2* and *Scope 3*. All scopes have defined the compensation handlers and the *Scope 1* has also failure handlers.

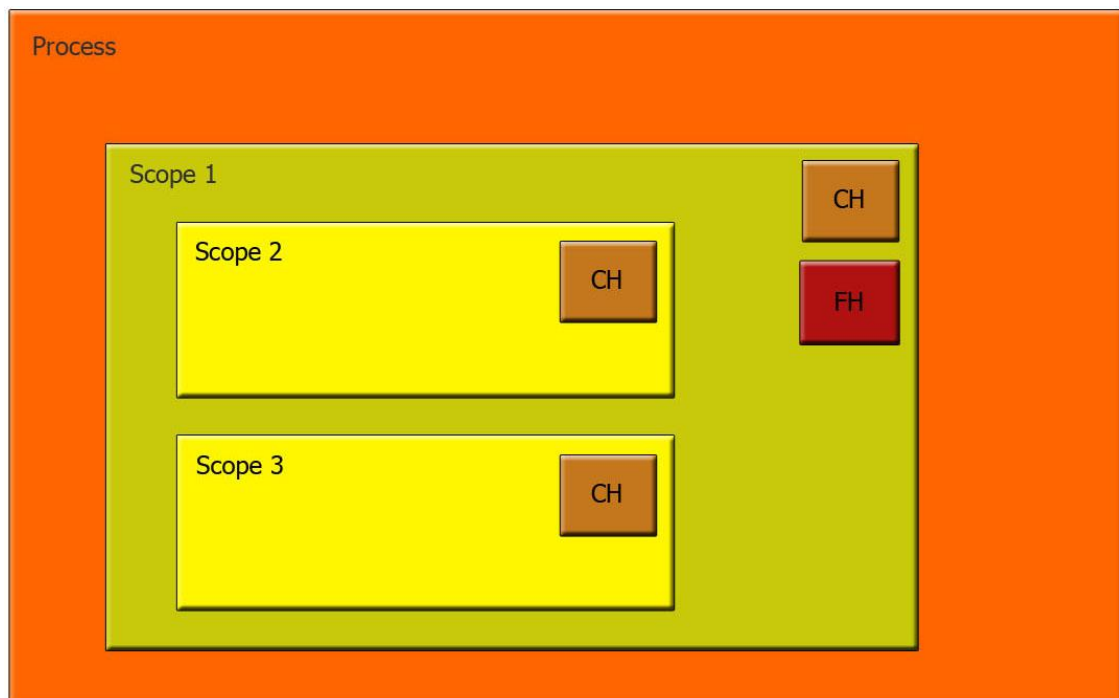


Figure 3.1 - Graphical representation of nested scopes

The process in order to treat failures, normally uses the scopes and the associated handlers. When a failure occurs within a scope and the scope have defined failure handlers, the failure will be caught and will be treated accordingly. A group of activities can be defined to treat specific failures, or it can be defined to treat all failures. Raising failures may lead the process to compensate previously concluded scopes. The definition of compensation handlers for

scopes will usually contain activities to undo what the scope did. The compensations can only be called or triggered when failure occurs. When compensation needs to be executed, it will run the compensations for the previous scopes that terminated. Returning to Figure 3.1., if a failure occurs in *Scope 3*, then the compensation for *Scope 2* must be executed. If *Scope 2* and *Scope 3* did not have compensation handlers, the process would use the compensation of *Scope 1*.

### **3.3 Recovery Mechanisms**

The recovery mechanisms present in WS-BPEL uses fault, compensation and termination handlers defined by the creators of the business process. Fault and termination handlers can be defined for the process itself or can be defined independently in each scope of the process. Error handling in WS-BPEL uses the concept of compensation defined in sagas. It attempts to reverse the affects of previous activities, that are part of large unit and needs to terminate for some reason. Logic work units are divided in scopes and for each scope it can be defined a work unit that contains the compensation instructions. Once all activities inside a scope are completed successfully, the scope can be compensated if it is required later. If a fault occurs or a fault is thrown, the fault handlers will execute the activities associated with the fault and then call the compensation of the scopes previously run. The compensations are run in the reverse order they were executed, so the compensation of the first scope executed will be the last to be invoked.

### **3.4 Analysis of Compensation Mechanisms**

Greenfield *et al.* in [12] focuses the shortcomings of the compensation methods applied to business processes. It is estimated that nearly 80% of the time used in the development of business process is to handle exceptions. This rate is very high because of the variables involved like: human interaction, network and concurrency. For instance, failures can occur even while handling other failures, concurrent business processes may be affected by shared



resources or failures that are not local to one party, but rather in the way peer processes interact.

The standard approach to dealing with failures and cancelation requests is based on ACID transactions, Saga's compensation transactions and exception handling derived from programming languages. The problem is the fundamental assumption of the standard approach, that all completed activity can be semantically undone. It is assumed that is possible to define the right compensation for all the activities. In WS-BPEL, an empty compensator is associated to activities that cannot be undone, making the enclosing scopes unaware of the incompleteness of the activity. Another flaw concerns the assumption that fault-handling should terminate all activities of the scope where the fault was raised. It makes sense in object-oriented programming languages, but in business process sometimes it is necessary to evaluate the current state of the scope and try to achieve a stable state. It is not possible also to create a customized handler, like for example, run a compensator for just one sub-activity and not the others of the scope.

Greenfield *et al.* also proposes the idea of an infrastructure to allow developers to define business application that maintain state and data consistently. This infrastructure should have:

- A language to express consistency conditions;
- A language to express systems design, treating cancelation and failure as events, just like a message arrival;
- Tools to check when the system maintains consistency;

### **3.5 Other Business Specifications**

There are many different tools that can use the WS-BPEL specification to provide the control between different web services.

### 3.5.1 BizTalk

BizTalk Server [13] is property of Microsoft and is currently in the 2009 version. This program serves mainly as a routing and manage service for messages between several partners, but has many others features. It is an integration base on which web services can be build. Since BizTalk is a WS-BPEL compliant system, it can be used to define Orchestrations. Orchestrations are processes that contain the rules to manage the business process.

Sagas are applied to relational database systems, but BizTalk Server extends similar concepts to the context of automated business processes. It is used within the orchestrations to provide a good support for handling external and internal data. The basic compensation model used is an extended version of saga's backward recovery. In this model, long-running transaction (LRT) is broadly equivalent to a saga. It can contain nested atomic transactions and each transaction is associated with a scope. Each scope can have a compensation block that contains the orchestration code used by the recovery system. If any nested transaction throws an exception, the long running transaction can invoke *backward recovery*. The invocation of default compensation is not automatic. It is always invoked from within the context of an exception handler on the LRT. So compensation can only be invoked if the exception is caught by the outer LRT.

There are also two forward recovery mechanisms. Retrying atomic scopes and resume suspended orchestrations. The first mechanism can only be used if the commit of a transaction fails and if the atomic transaction scope throws an instance of *RetryTransactionException*. It can perform up to 21 retries, if it still fails, BizTalk will suspend the orchestration instance. When an orchestration is suspended, it can be resumed manually or using a custom script. Resuming the orchestration consists on re-starting the process from the most recent persistence point that is the point when a scope has committed all operations.

### 3.5.2 WebSphere

WebSphere [14] are a group of products by IBM that are developed over open standards like Java [4] and XML. Within this group there are a few that work with business processes like for instance Business Process Choreographer (BPC) [15]. WebSphere, like BizTalk is also WS-BPEL complaint, so it can also process de WS-BPEL syntax within its tools.

BPC has two types of processes: *long-running processes* and *microflows*. Long-running processes consists of several chained transactions. This process is interruptible and can have duration between hours and even years. A microflow is a short-lived process that runs inside a unit of work and has a maximum duration. The activities within a microflow are automatic and cannot wait for inbound events once it is started.

BPC also uses the notion of compensation to treat failures, but the implementation depends on the type of process. Microflow must be compensated as a whole and long-running may be compensated partially.

### **3.6 Summary**

This chapter provided an insight on the language used to describe a business process. It can use simple activities or complex ones, mixing it all. Normally these activities are grouped inside scopes, usually when they are related with each others. Scopes can have compensation and failure handlers associated with them. Also was showed the recovery mechanisms present in the business process and other tools that use the WS-BPEL and add a few more options to those recovery mechanisms.

The next chapter will provide a formalization for business process.

## **4. WS-BPEL Formalization**

The WS-BPEL can be formalized using different approaches. This thesis will focus formal languages with graphical notation to provide a better understanding of the business process and what it can do.

### **4.1 Formal Languages with Graphical notation**

The formal languages that are present here, all have graphical notation and a two of them already were used to map WS-BPEL.

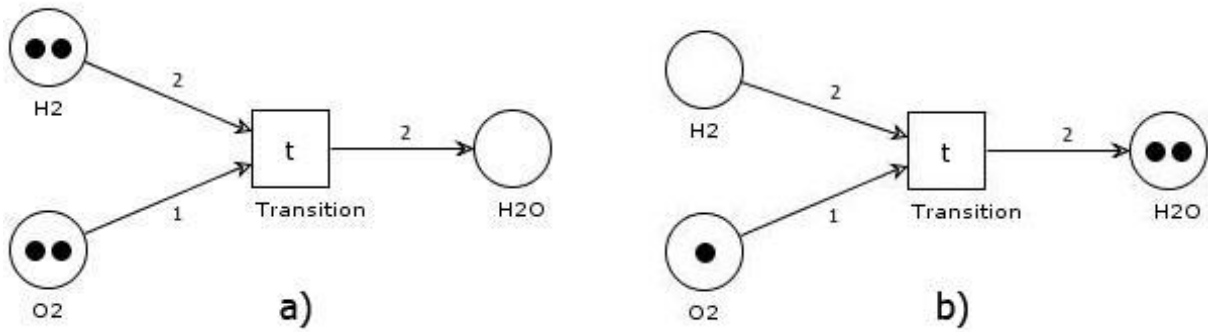
#### **4.1.1 Petri Nets**

Petri Nets are a graphical and mathematical modeling tool. It is used for describing and studying information processing systems that can be concurrent, distributed, and parallel among others. It is possible to set up state equations, algebraic equations and other mathematical models that define systems behavior. The concept of Petri Net was developed by Carl Adam Petri's dissertation dated 1962 and has been evolving ever since. Petri Nets can be extended to formalize many types of systems, and have become one of the most favorite graphical formal languages [16].

Petri Net has two types of nodes: places and transitions. The nodes are connected using directed arcs. This connection must be between different types of nodes. If a place is the source of an arc, it is called input place. If it is the destination of an arc, it is an output place. Each arc can have different weight which can consume or supply tokens depending on its connections. Tokens are non negative integer that refers to a number of data items or resources available. The presence of tokens in a place is called marking and a transition is enabled if each input place has tokens. Only an enabled transition may be fired, and if it is

fired, the tokens will pass from the input place to the output place. Graphically, places, transitions, arcs, and tokens are represented respectively by circles, bars, arrows, and dots.

A Petri Net transition is exemplified by this Figure 4.1. This illustration shows a chemical reaction. When transition  $t$  is fired, the marking will change from a) to b) consuming 2 tokens from the input place  $H_2$  and 1 from input place  $O_2$ , because that is the weight of the arcs. That transition will then supply 2 tokens into an output place  $H_2O$ .



**Figure 4.1 - Illustration of a Petri Net firing rule**

Among the works done with Petri Nets, there are some concerning the conversion WS-BPEL syntax to Petri Nets, which includes the compensation mechanisms. Among these are the work of Stahl [17], Lohmann [18, 19], König [20] and Ouyang [21].

#### **4.1.2 Statecharts**

*Statecharts* diagrams were developed by David Harel [22] as an extension of state machines and state diagrams to specify and design complex discrete-events systems. These diagrams are a graphical notation and extend the normal state diagrams with 3 notions: hierarchy, concurrency and communication. Those notions will allow the creation of simple diagrams that can illustrate complex behaviors.

The two main components of *Statecharts* are the states and transitions. There are three types of states: basic states, and-states and or-states. The or-states are sequential sub-states, the and-states are concurrent sub-states and the basic states have no sub-states. The transitions are

events between states. The transactions are composed by: source state, target state, event, action and condition.

The Figure 4.2 is a *Statechart* example in [22]. In this example,  $D$  is a state that can have the  $A$  or  $C$  states active, but not both. If an event  $\gamma$  occurs in state  $A$  transfers the system to state  $C$ , but only if condition  $P$  holds at the instant of occurrence. The event  $\beta$  takes the system to  $B$  from either  $A$  or  $C$ . Event  $\alpha$  and  $\delta$  transfers the system from  $B$  to  $A$  or  $C$  respectively.

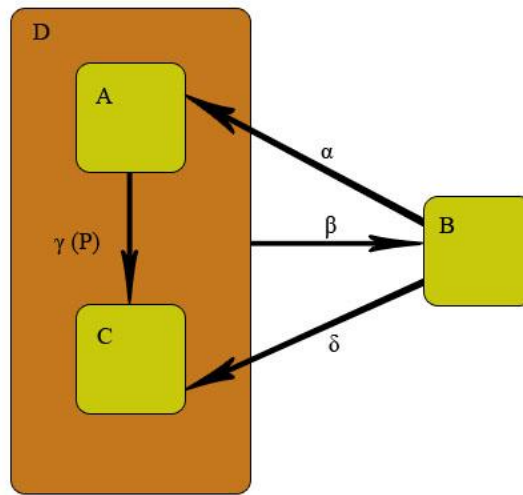


Figure 4.2 - Statechart example

#### 4.1.3 Finite State Process

Finite State Process (FSP) is a textual notation much like a process calculus [23] by Magee, Kramer, *et al* [24]. It is designed to be machine readable and is used for specifying concurrent programs. Once the FSP is created, it can be used within a modeling tool, for instance, the Labeled Transition System Analyzer (LTSA) [24] that compiles the FSP into a graphical workflow process.

FSP has several operators defined in its semantics. Some of the operators are presented in Table 3. A thesis done by Howard Foster [25] models WS-BPEL notation into FSP, but does not fully cover the compensation mechanisms. This work is showed later in the Section 4.2.2.

Name	Operator	Example	Description
Action Prefix	->	x->P	an action x is engaged and then the process P is executed;
Choice		x->P y->Q	action x or action y are engaged and then the process associated with each one is executed;
Recursion			the behavior of a process may be in terms of itself;
End State	END		it appears when a process terminates successfully and has no more actions;
Sequential composition	;	P;Q	Describes a process P that when it reaches the end state, starts the process Q;
Parallel composition		P  Q	Describes that both process can be executed in parallel.

**Table 3 - Some FSP operators**

## 4.2 Related Work

This Section covers different approaches used to formalize the WS-BPEL into a graphical notation. It presents works done in Petri Nets, Finite State Process and Workflow Nets.

### 4.2.1 Petri Nets

Like FSP, also Petri Nets have been used to model business processes. In [17] Stahl presents a pattern-based Petri Net semantics for WS-BPEL. It covers the standard behavior of WS-BPEL and includes also faults, events and compensation. Although WS-BPEL being a textual language, it does not have formal methods that would help its verification. Therefore a formal semantic is needed to resolve ambiguities and inconsistencies. Usually the existing formal languages used for WS-BPEL covers the standard behavior, but does not support fault handling or compensation.

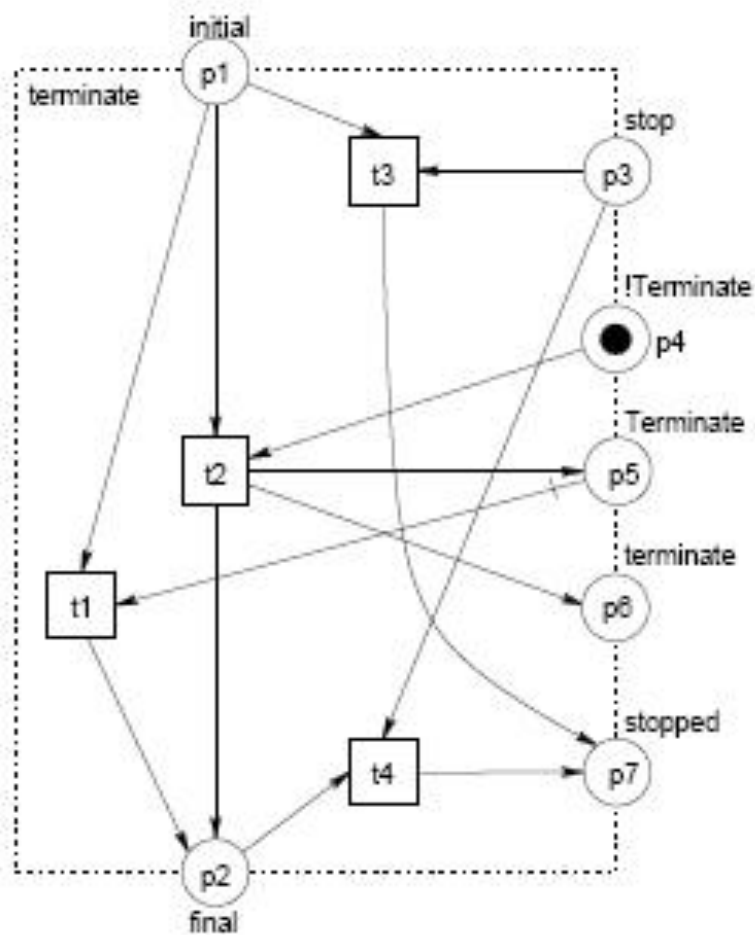
The goal of Stahl's work is to translate every WS-BPEL process into a Petri Net. The WS-BPEL constructors are translated to Petri Net, creating a pattern. Each pattern has an interface to join other patterns, can have parameters and carry several inner constructs as WS-BPEL, keeping all its properties. That collection of patterns is the Petri Net semantics of this work.

Some design decisions were made when translating special concepts of WS-BPEL. Positive control is the flow from top to bottom and communications between processes flow horizontally. In order to stop positive control, every activity pattern was extended by a stop component that is called when a scope need to be stopped. It was also needed to save all executed scopes, because when an implicated compensation handler was invoked, all the compensation handlers of its child scopes needed to be invoked as well.

A tool was developed to automatically transform WS-BPEL processes into Petri Nets. Currently it cannot be used with High Level Petri Nets, which are nets that extended the normal behavior using color, time or hierarchy.

One of the examples showed by Stahl is the terminate activity. It is executed to terminate the whole process instance. In Figure 4.3, the process state changes to terminated and the stop pattern is use to end the process. Two things can happen: the process state is already terminated (t1) or the termination of the process is started by changing the state to terminated (t2). The Stop Patterns are made by transitions t3 and t4.





**Figure 4.3 – Terminate Activity in Petri Net**

Another work by Lohmann [18] also presents an extension of a Petri Net semantics for WS-BPEL, but also covers the latest version, WS-BPEL 2.0 specification. It uses Open Workflow Nets (oWFNs) which are a special class of Petri Nets. The oWFNs have a simple formal basis to model services and interactions, preserving the same properties associated with Petri Nets. These oWFNs were implemented in a compiler (BPEL2oWFN) [3]. Like in [17], each construct of WS-BPEL can be translated into a Petri Net, creating a pattern. Patterns can be connected to each other by interfaces forming a WS-BPEL structured activity.

Lohmann created a more compact model by simplifying and reducing some aspects as dead-path-elimination and the <scope> pattern. This compact model changed some graphical notations present in [17], including the use of color. Dashed place is a copy of place with the same label or read arcs are unfolded to loops. Control flow can be stopped at yellow places,

and a fault can be thrown on orange places. Blue transitions access variable places, among others. Figure 4.4 comes as an example of this notation.

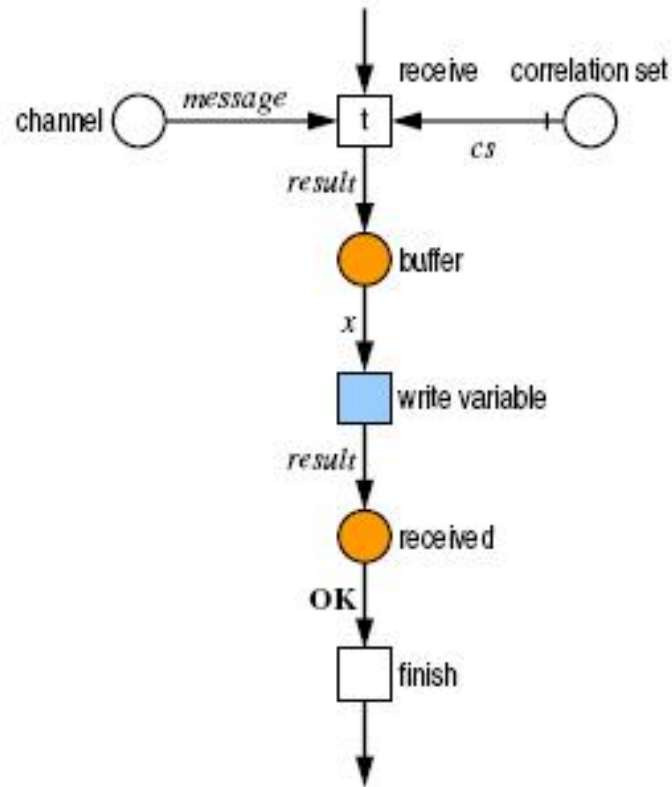


Figure 4.4 – Pattern “receiving a message”

The semantics cover all data and control flow aspects of WS-BPEL, but does not cover the instantiation of process instances and message correlation. Future work will involve semantics that cover all the lifecycle of process instances.

#### 4.2.2 Finite State Process

Finite State Process (FSP) has been used to model business process, like for instance in Foster’s thesis [25]. According to Foster, “The main objective of this work is to provide a rigorous approach to specifying, modeling, verifying and validating the behavior of web service compositions with the goal of simplifying the task of designing coordinated distributed services and their interaction requirements.“. It presents a guide to model BPEL4WS semantics in FSP models and Labeled Transition Systems (LTS). The diagrams

provided by the LTS aids in the comprehension of the most elaborated operations executed by WS-BPEL, like for instance, concurrent processes. These diagrams are provided by a tool that converts the FSP in a LTS. In order to process the WS-BPEL into a FSP, a plug-in was developed for that tool.

The modeling WS-BPEL in FSP made some assumptions and has some limitations. Foster assumes that a process starts at the first receive activity specified in the process, because multiple start points would affect the order of the activities. There is no implementation of synchronization between events, like the interaction of clients in long-running process. The mapping is limited in the translation of variables and does not include event handling as part of an activity scope. And it can only model the behavior of a single process.

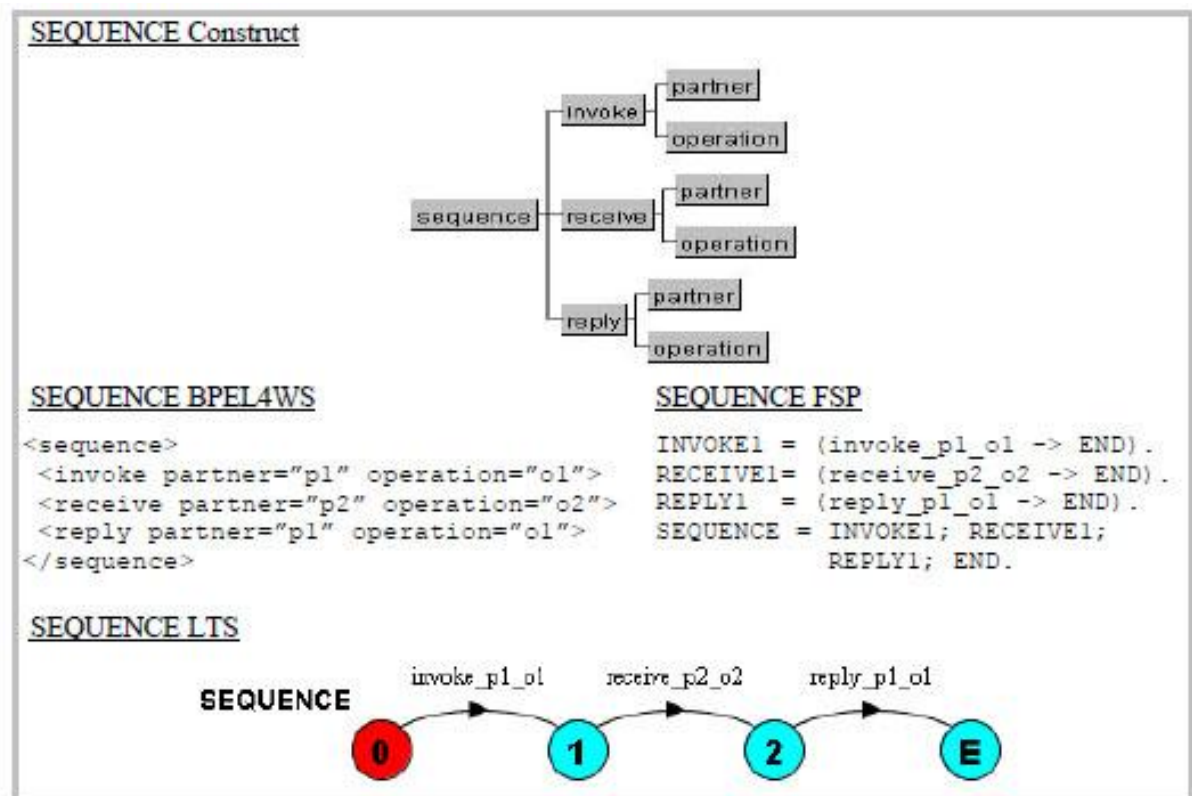


Figure 4.5 – Sequence example

As an example of the modeling involved in Foster thesis, Figure 4.5 shows the correlation between a sequence activity in WS-BPEL, FSP and LTS. The sequence scope in WS-BPEL

provides the order in which the activities should be executed. In this case, a partner invokes a service, then receives a message from that partner and replies, ending the sequence. In FSP it is represented by a sequence composition where the activities are separated by “;” operator. In LTS the start activity has the red background and the last activity is labeled with “E”. These activities are connected by arrows indicating the order in which they are going to be executed.

### 4.2.3 Workflow Nets

Workflow nets (WF-net) were introduced by Wil van der Aalst [26] [27] and are an extension of the Petri Nets. Workflows, in a business process, are the tasks that are needed to be executed and their order. The transition proposed by Aalst, is that tasks are modeled by transitions, conditions are modeled by places and cases are modeled by tokens. Van der Aalst introduces also conditional blocks (represented on Figure 4.6) and the use of triggers (shown on Figure 4.7) creating a simple notation to implement a workflow based on the Petri Nets. Any WF-net must satisfy two requirements: every net must have a source place and a sink place which represents the start and finish place of the net and every transition and place must be in a path between these two places.

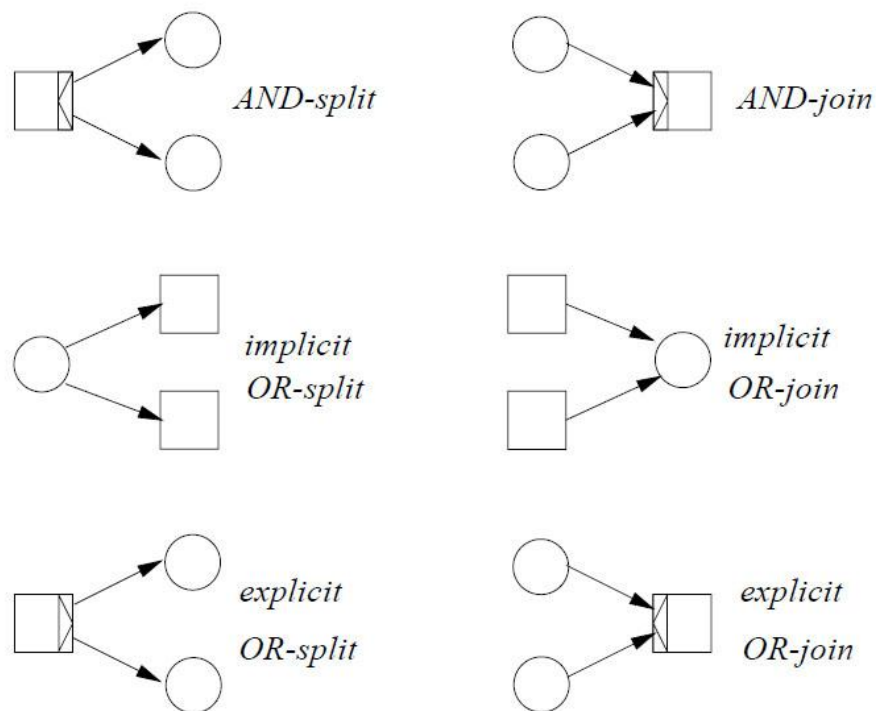
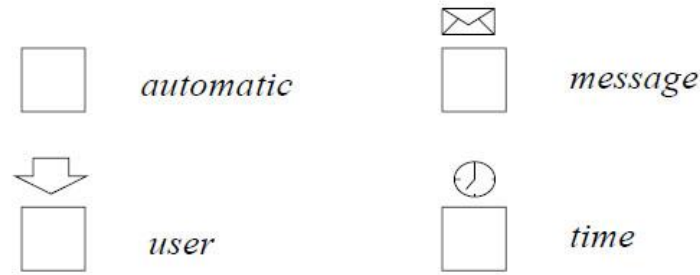


Figure 4.6 - Workflow condition blocks



**Figure 4.7 - Workflow triggers notation**

#### **4.2.3.1 Workflow Petri Net Designer**

Workflow Petri Net Designer [28] (WoPeD) is a tool to model, simulate and analyze workflow processes using the workflow nets introduced previously. It is an open-source software developed at the Cooperative State University Karlsruhe and is currently in a version 2.3.1. It allows the use of workflow components to build nets that can generate WS-BPEL code. Each component can have different configurations to express basic or complex activities which includes, for instance, the declaration of variables. The tool supports reach ability testing, deadlocks and soundness analysis. It also includes a token game to see how the net evolves depending of the conditions imposed. Some images of the application are shown by Figure 4.8, Figure 4.9 and Figure 4.10.

Figure 4.8 shows the available definitions for a transition. It allows defining the type of transition and what triggers it. It can also include the time it takes to perform and its role. Different components have different definitions.

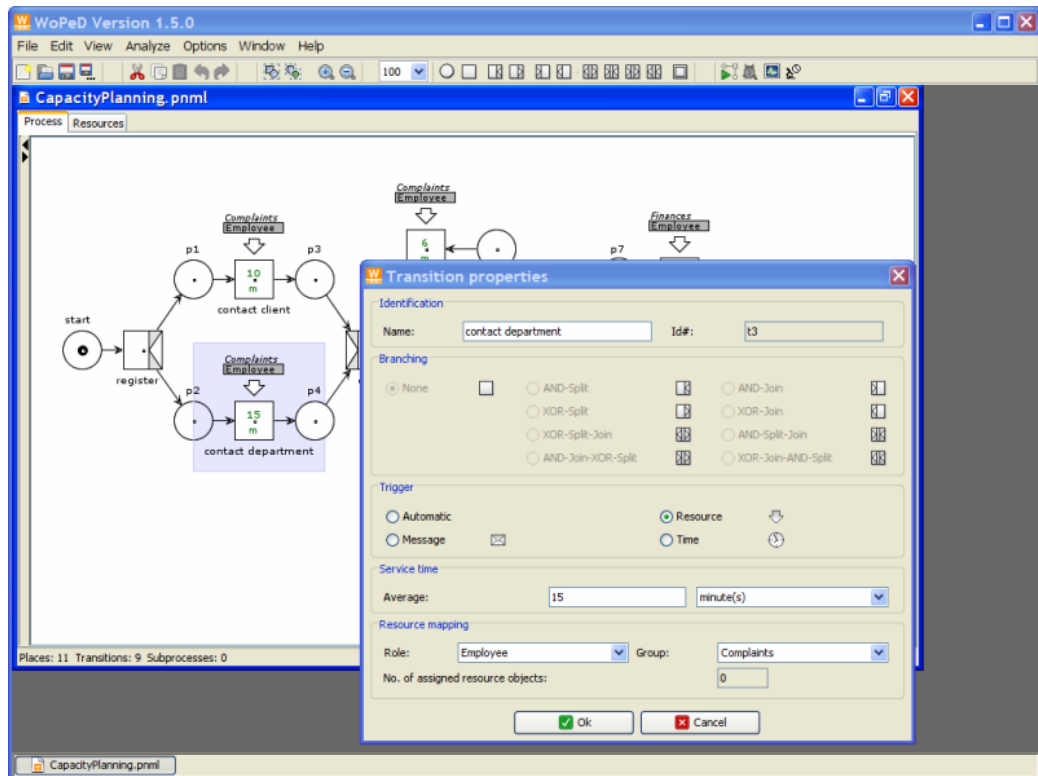


Figure 4.8 - WoPeD screenshot of property editor

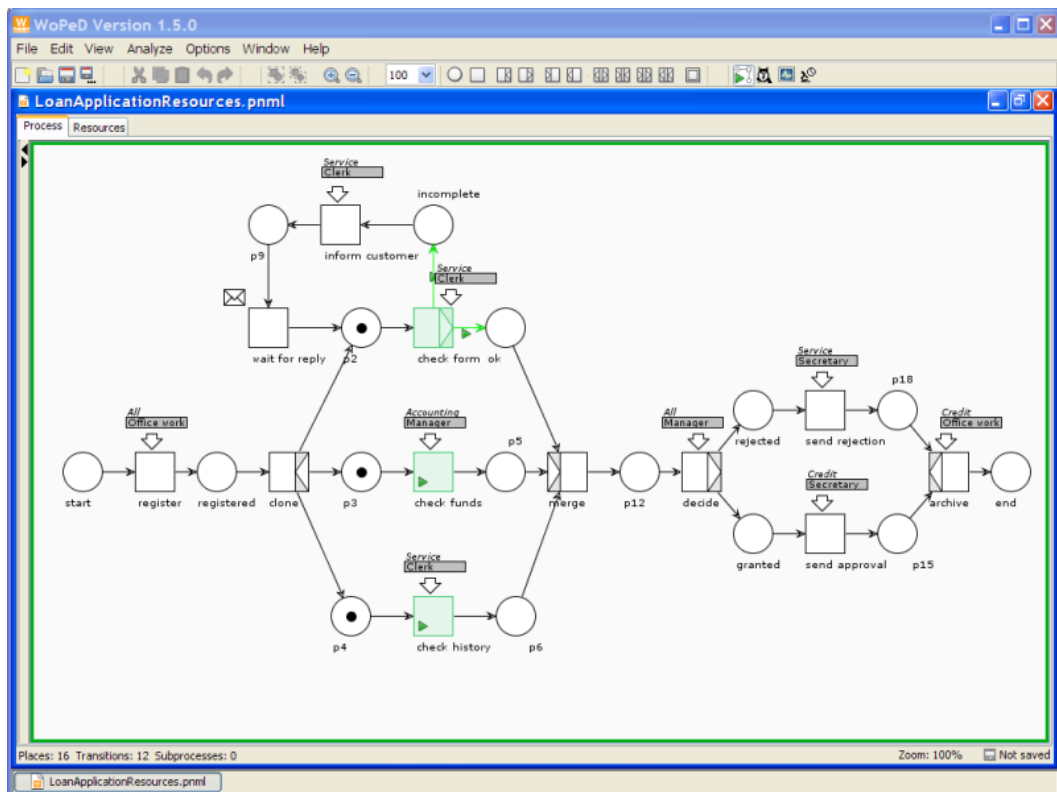
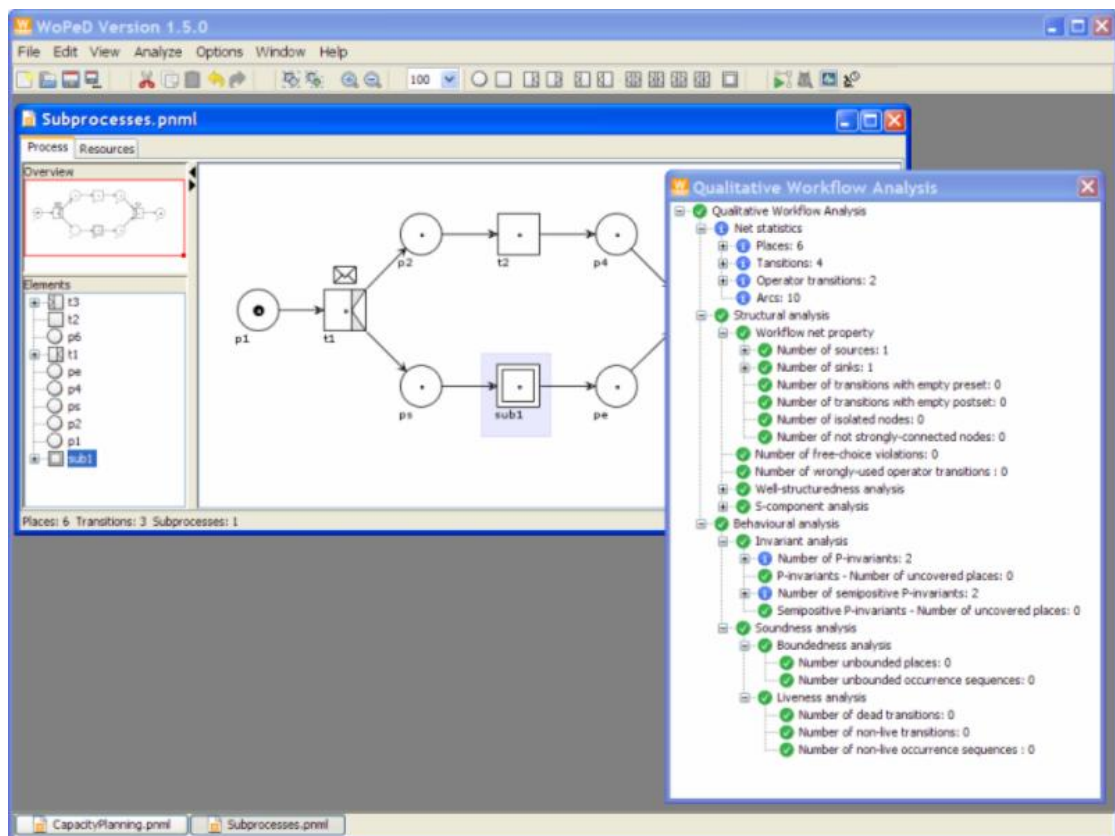


Figure 4.9 - WoPeD screenshot of token game

Figure 4.9 exemplifies a token game. A token game allows the user to see the available paths, and choices that the implemented Net has. First one must assign tokens to the places that needs it. At least one must be assign to the starting place in order for it to work. In no path exists between the starting and finish place, then the token game does not start. Assuming normal behavior, when a choice is presented to the user, a small play sign appears and the choice made by the user will lead to the next possible choices. This will be done until reaching the end of the net.



**Figure 4.10 - WoPeD screenshot of soundness analysis**

Figure 4.10 shows the properties of the displayed net, including if it has soundness and liveness.

### **4.3 Summary**

This chapter focused different alternatives to provide a formal interpretation to WS-BPEL by using graphical notation. This included Finite State Process, Statesharts and different types of Petri Nets which are the most common formal languages used when it concerns WS-BPEL. There are several works using low level and high level Petri Nets to map WS-BPEL components but do not focus recovery mechanisms in detail. The next chapter maps the WS-BPEL language into the chosen formal language, implementing patterns for the activities presented in section 3.2.














## 5. Mapping WS-BPEL to Workflow Petri Nets

In this chapter the WS-BPEL activities are mapped into Workflow Petri Nets (WPN). WPN graphical notation available in the WoPeD tool is simple to follow and allows the creation of sub-processes. The WoPeD can already be used to generate WS-BPEL code, but visually this is not perceptive. So in order to become more clearer, the WoPeD components will be used to form patterns that represent the activities available in WS-BPEL. These patterns can be united replacing the finish place of one pattern by the start place of the next pattern.

### 5.1 WPN Components

Here is presented the components that are going to be used to map WS-BPEL into WPN, shown in Table 4. The nets created follow the same principles of the Petri Nets. A place cannot be connected to another place, nor the transitions to other transitions. A net must start and end in a place.

WPN Component	Description
 Place	Represents a certain state of the workflow
 Transition	Represents the event that allows the states to change

 SubProcess	A sub Petri Net, in order to manage complexity
 Transition AND-Join	Closing point of an parallel branching
 Transition XOR-Join	Closing point of an alternative branching
 Transition AND-Split	Starting point of an parallel branching
 Transition XOR-Split	Starting point of an alternative branching
 Transition AND-Split-Join	Closing and starting points of parallel branching
 Transition XOR-Split-Join	Closing and starting points of alternative branching
 Transition AND-Join XOR-Split	Closing point of parallel branching and starting point of alternative branching
 Transition XOR-Join AND-Split	Closing point of alternative branching and starting point of parallel branching

**Table 4 - WPN components**

## 5.2 Basic Activities

The basic activities presented in section 5.1 will be mapped in this section, except the extension activity.

### Receive

Receive activity waits for a message from a designed partner. The mapping is shown in Figure 5.1.

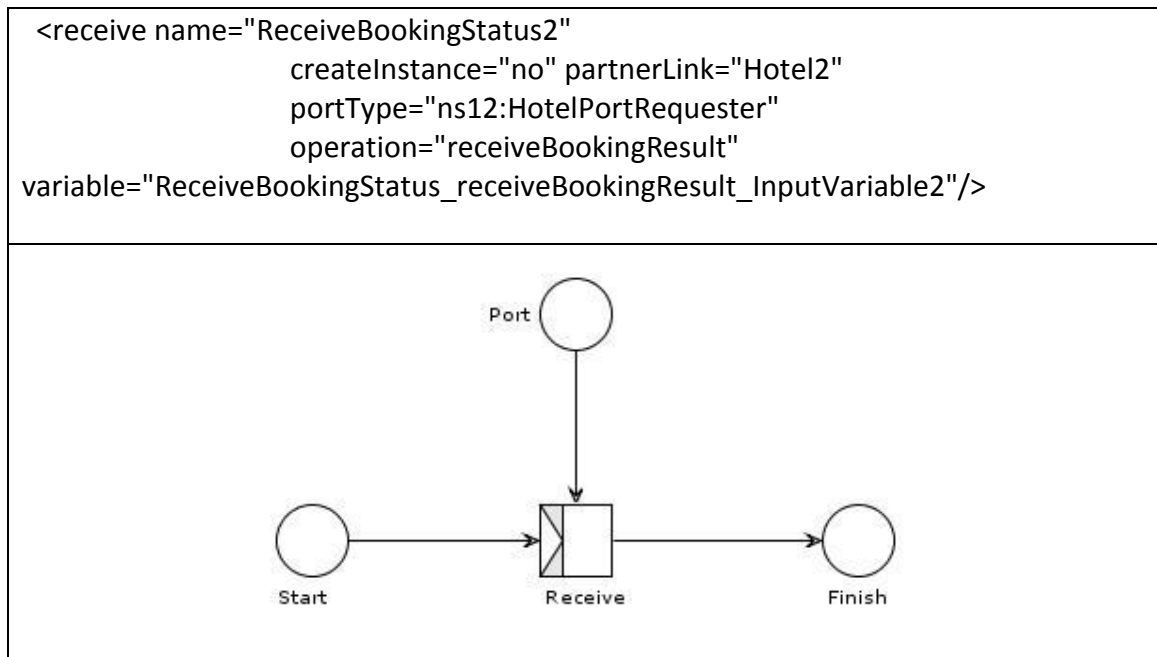


Figure 5.1 - WPN pattern for receive activity

The transition *Receive* waits for the port associated to be triggered with some message. Once a message arrives, the WPN can finish.

## Reply

Reply activity responds to an invocation by a partner. The mapping is shown in Figure 5.2.

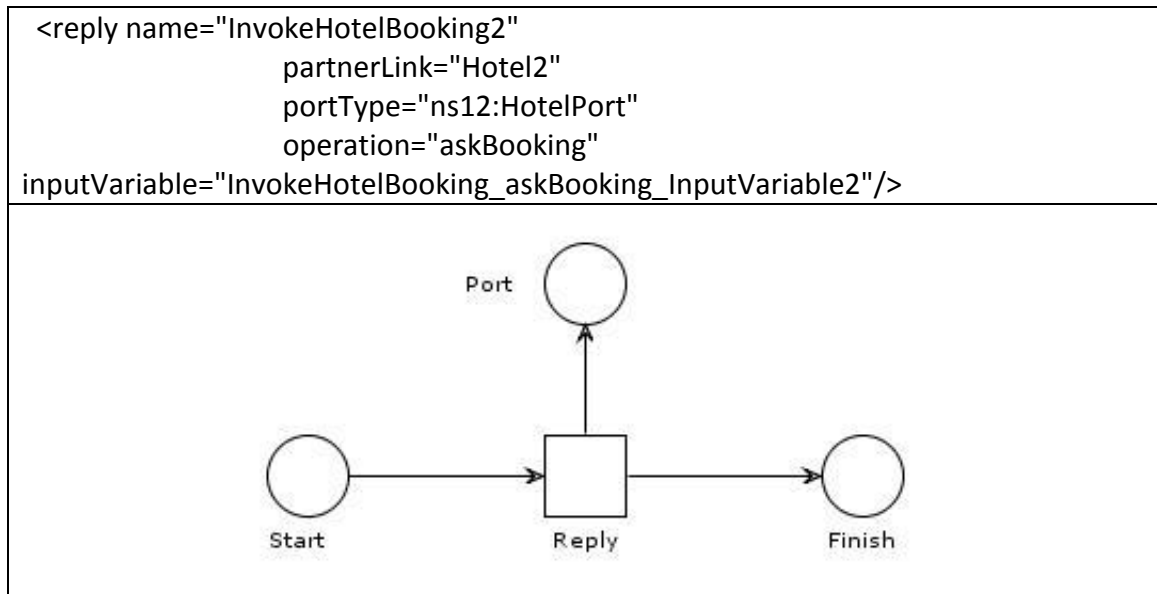


Figure 5.2 - WPN pattern for reply activity

The transition *Reply* sends a message to a specific port and finishes. This mapping assumes that it sends a message and finishes at the same time. This is inaccurate, but in order to keep the resulting diagram simple and not introduce more transitions it is assumed that they are simultaneous.

## Invoke

Invoke activity send a message to a partner to invoke an operation, shown in Figure 5.3.

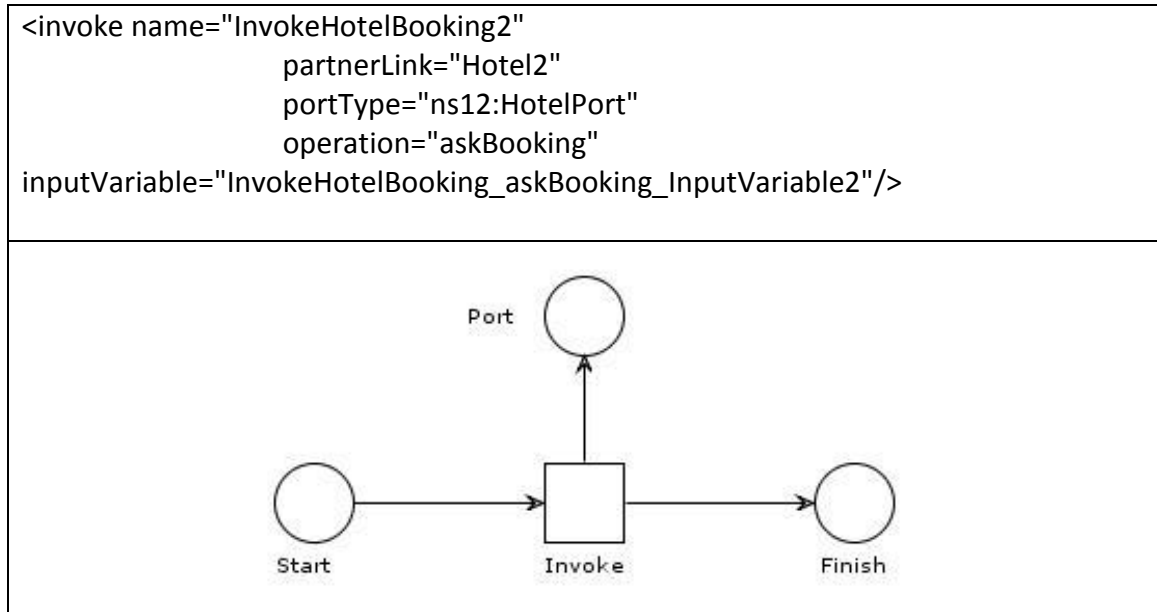


Figure 5.3 - WPN pattern for invoke activity

Similar to *Reply*, the *Invoke* transition sends a message to a partner and continues with the remaining activities.

## Assign

This activity exchanges data between variables. Shown in Figure 5.4.

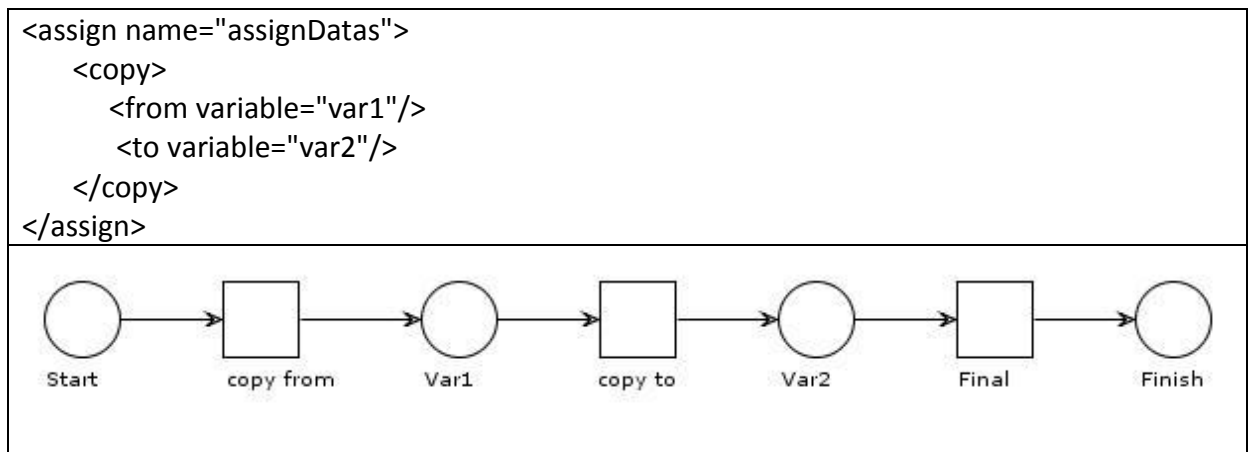


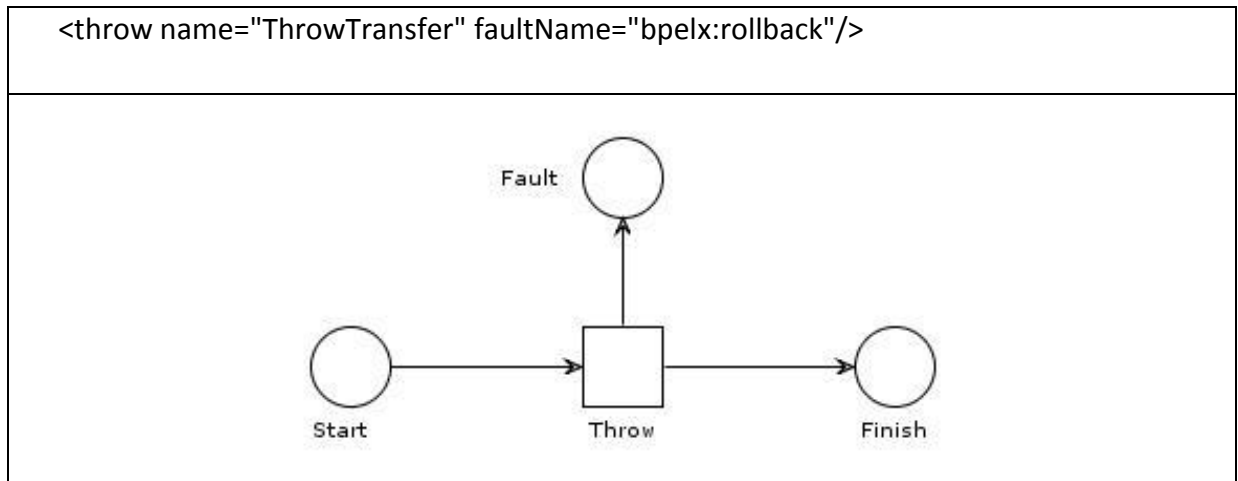
Figure 5.4 - WPN pattern for assign activity

A transaction copies a variable, or element of an variable *Var1* to other transaction that puts the value into *Var2* and then finishes.

## Throw

This activity throws an exception to be catch by the failure handlers. The pattern is presented in

Figure 5.5.

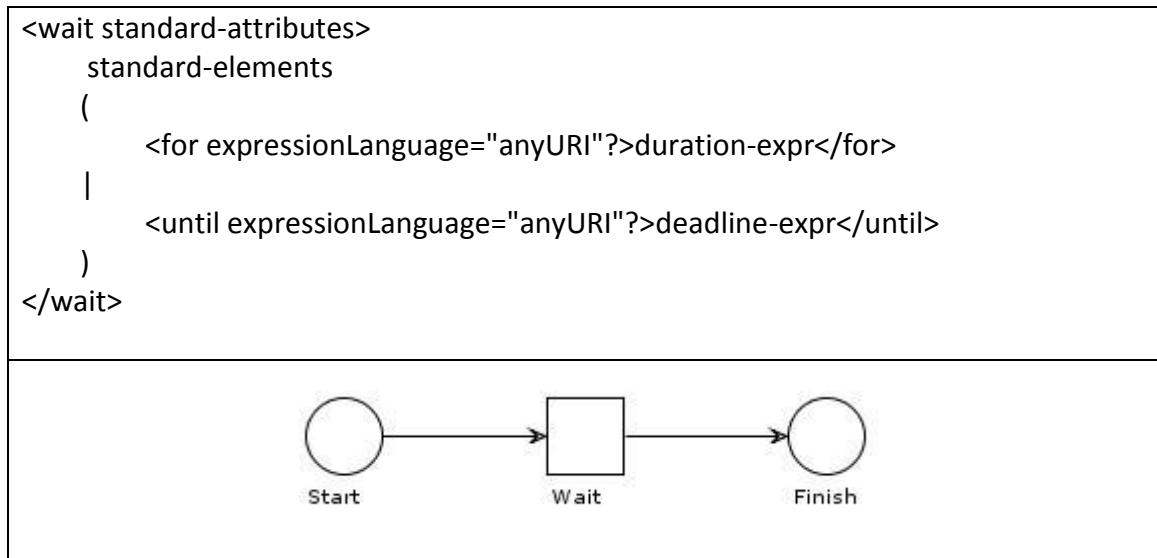


**Figure 5.5 - WPN pattern for throw activity**

The Throw representation is similar to the invoke representation. The difference is that the thrown fault will affect the rest of the activities, since it will be have to be threaten within the fault handlers.

## Wait

The Wait activity waits for a period of time or until a certain deadline.

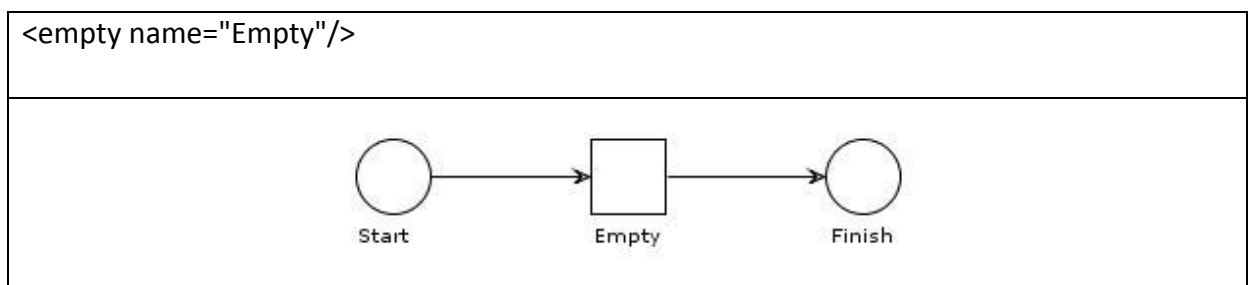


**Figure 5.6 - WPN pattern for wait activity**

Since the WPN does not take in consideration time, the WPN representation of Wait activity in Figure 5.6 is just the transition between Start and Finish.

## Empty

The Empty activity does not do anything.



**Figure 5.7 - WPN pattern for empty activity**

Empty transition just passes from the Start place to the Finish place shown in Figure 5.7.

## Exit

This activity ends the process without executing other activities.

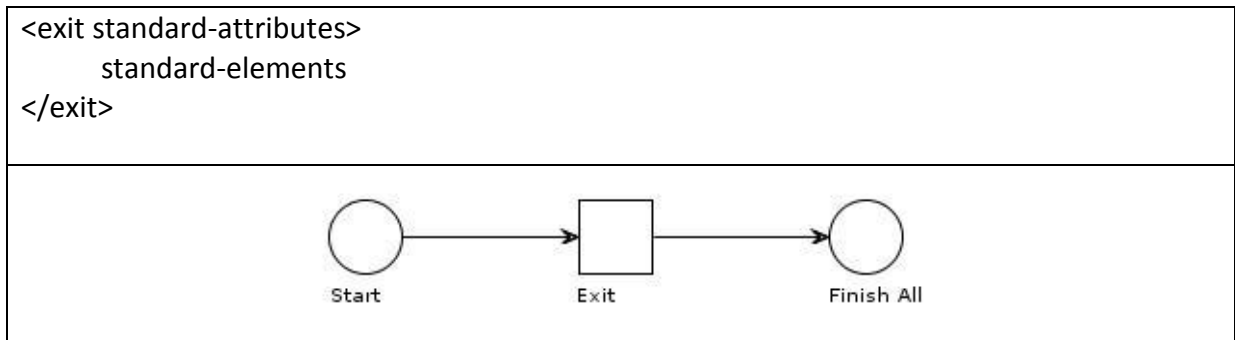


Figure 5.8 - WPN pattern for exit activity

In the exit mapping in Figure 5.8, the Finish place cannot be connected with other components.

## Rethrow

This activity does the same as the Throw activity but can only be used inside fault handlers.

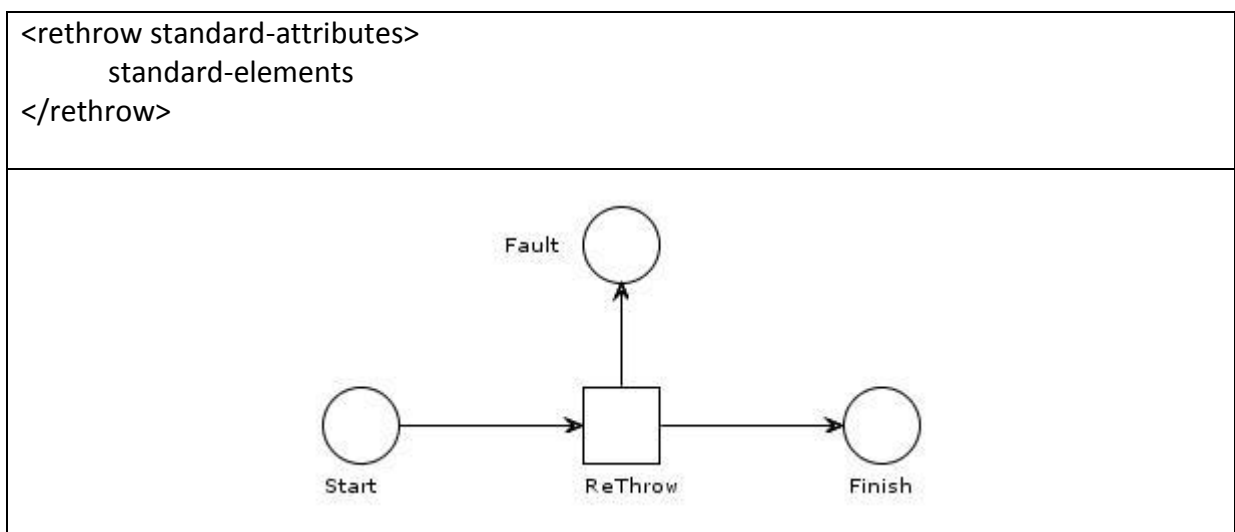


Figure 5.9 - WPN pattern for Rethrow activity

The *Rethrow* transition send a fault signal and finishes. This is mapped in Figure 5.9.



## Compensate

This activity can only be used within the failure, compensation and termination handlers. In Figure 5.10 the *Compensate* transition calls a specific compensation of a scope.

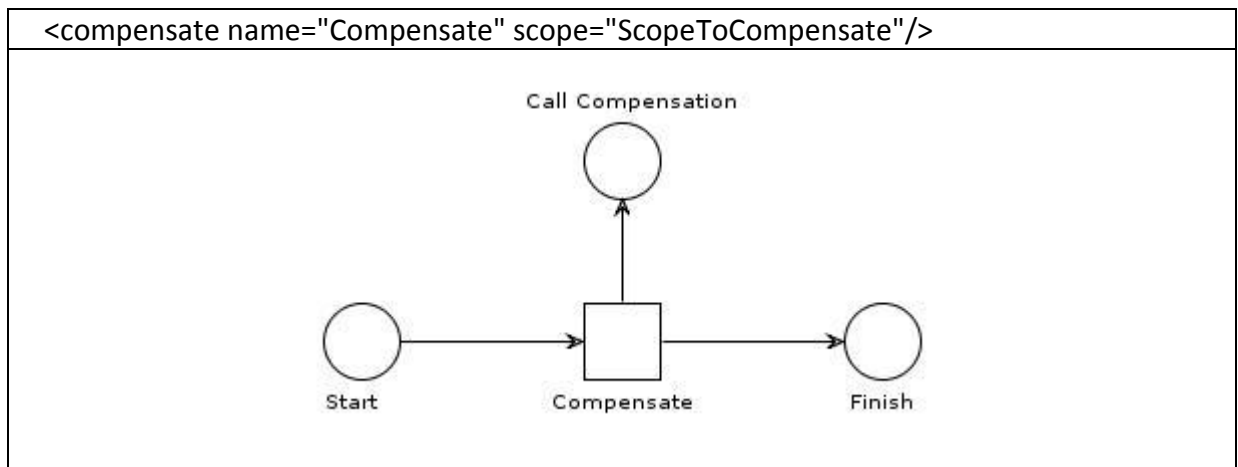


Figure 5.10 - WPN pattern for the compensate activity

## 5.3 Structured Activities

The Structured activities presented in section 3.2 will be mapped in this section. These patterns contain sub-nets that can be used to put other patterns, simple or complex.

### Sequence

Can contain multiple activities that are executed in sequence, illustrated in Figure 5.11.

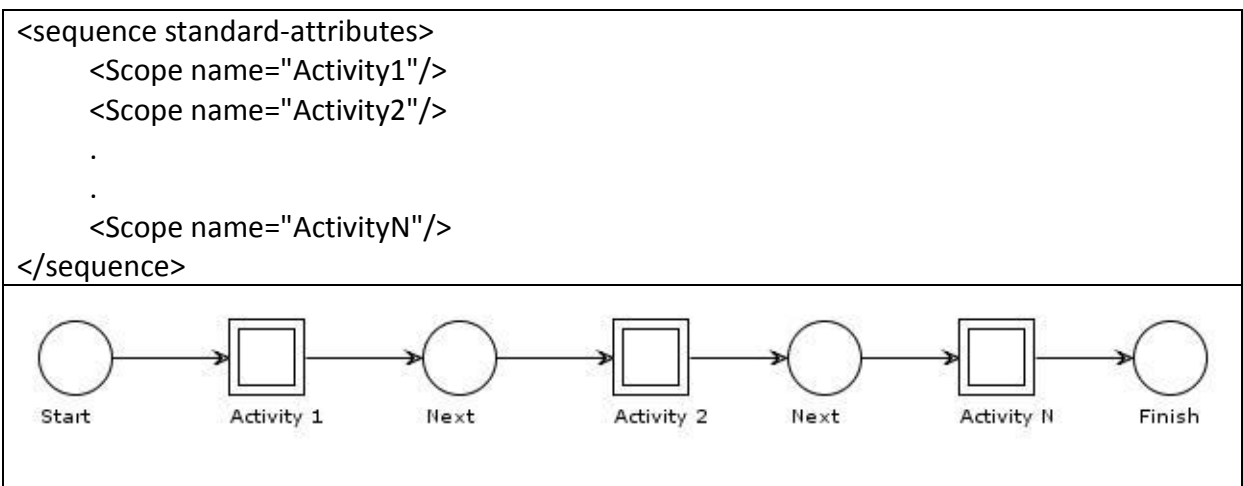


Figure 5.11 - WPN pattern for sequence activity

Sequence is a group of activities than are executed in a specific order. In this mapping the transitions are replaced by the sub process. The sub process can be changed to an explicit pattern, but doing so the *Start* and *Finish* places of the patterns must replace the *Next* places.

## If

Depending on the condition, it will execute de associated activity or activities.

```
<if>
  <condition expressionLanguage="anyURI"?>bool-expr</condition>
    Activity 1
  <elseif>
    <condition expressionLanguage="anyURI"?>bool-expr</condition>
      Activity 2
  </elseif>
  .
  .
  <else>
    Activity N
  </else>
</if>
```

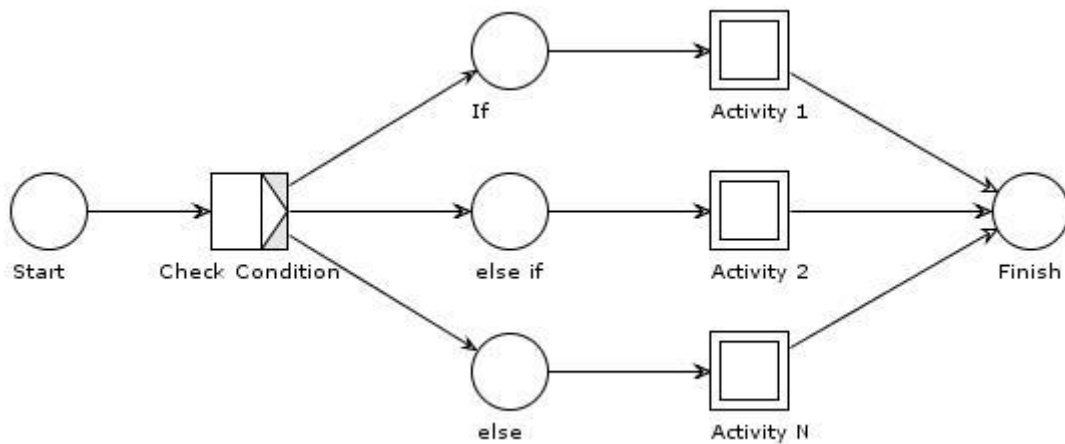


Figure 5.12- WPN pattern for if activity

Figure 5.12 shows the If activity mapping. The *Check Condition* transition check to which place it should go and executes the associated *Activity* and finishes.

## While

While a condition is true an activity is executed. The mapping is shown in Figure 5.13.

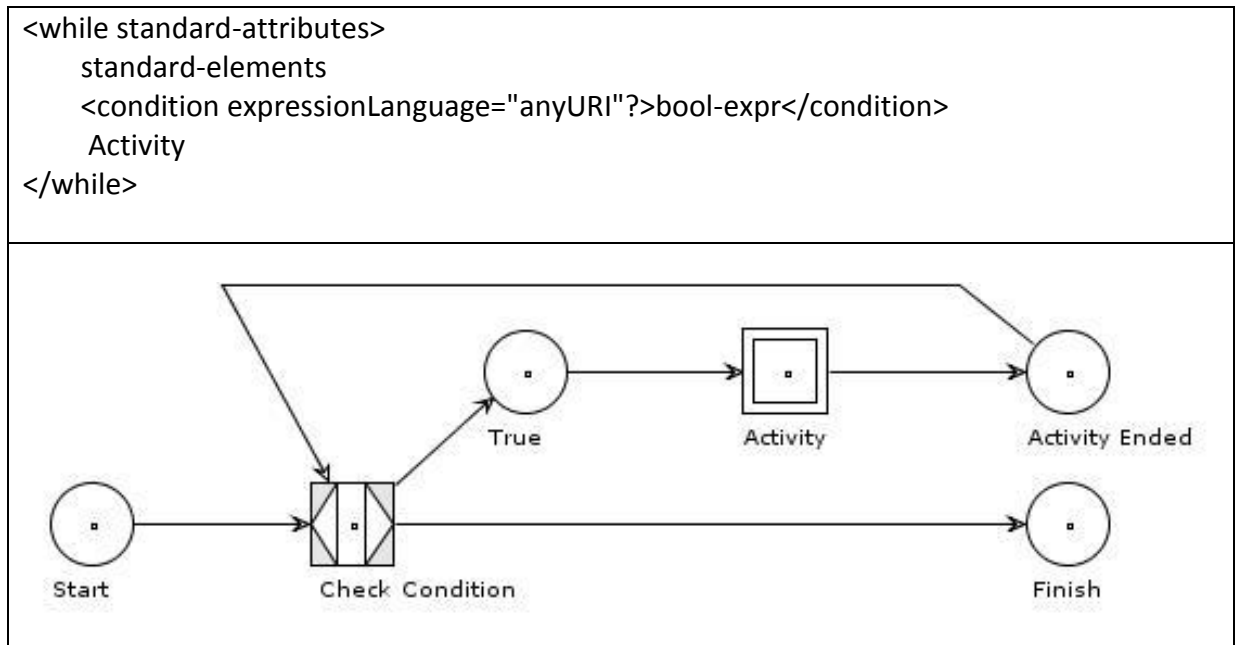


Figure 5.13 - WPN pattern for while activity

The while representation is composed by a transaction that verifies if a condition is verified. If it is true, a sub process *Activity* runs and in the end the condition is verified again. This will occur until the condition is false.

## Repeat Until

This activity is similar to the while activity but the condition is verified at the end.

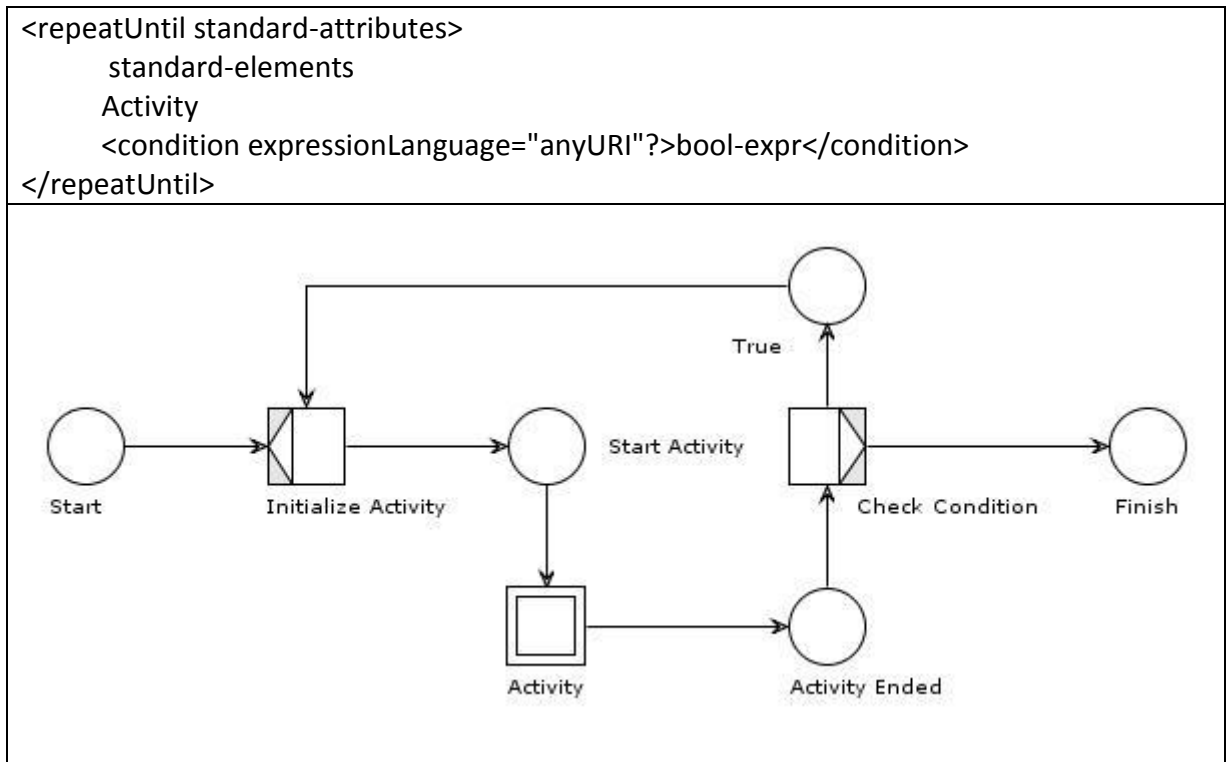


Figure 5.14- WPN pattern for repeat until activity

Repeat Until is shown in Figure 5.14. The verification of the condition is checked after the sub-process *Activity*. So the activity will be executed at least one time.

## Pick

Waits for a specific message to arrive and the mapping for this activity is shown in Figure 5.15.

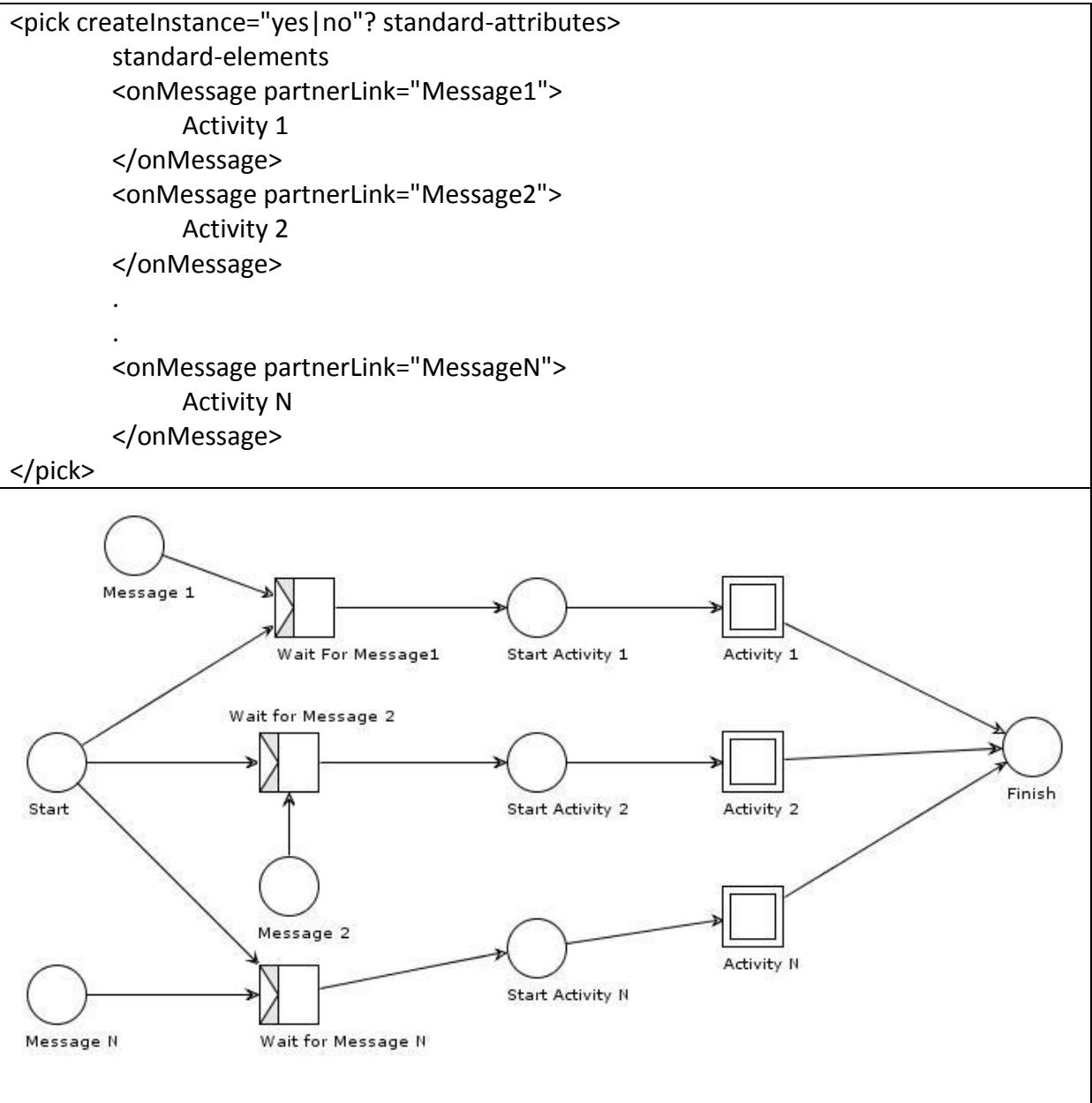


Figure 5.15 - WPN pattern for pick activity

This pattern is composed by a N number of transitions that are related to a specific message.

Once a place with a message is activated, the associated activity starts and finishes. The others do not run.

## Flow

Allows concurrent Activities.

```
<flow standard-attributes>
  standard-elements
  <links>
    <link name="LinkFlow1" />
    <link name="LinkFlow2" />
    .
    .
    <link name="LinkFlowN" />
  </links>
  Activity 1
  Activity 2
  .
  .
  Activity N
</flow>
```

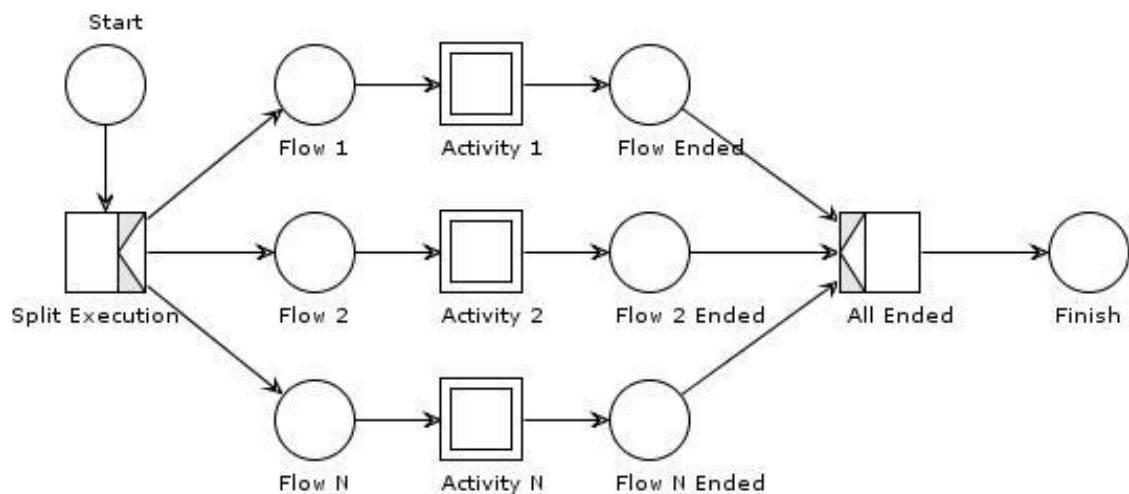
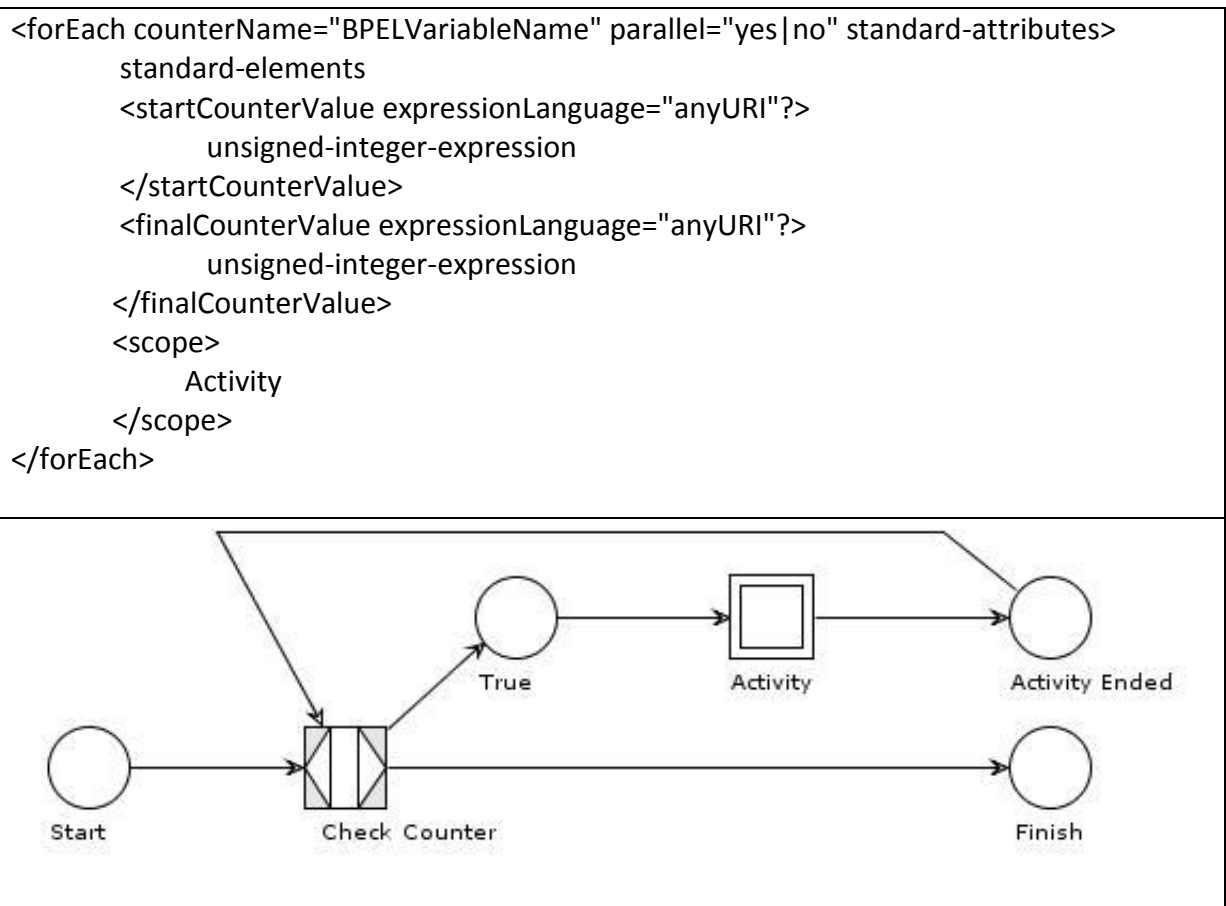


Figure 5.16 - WPN pattern for flow activity

This Pattern allow N Flows to run concurrently executing the Activities. The flows can start by any order. Once all activities are finished, the pattern ends. Figure 5.16 presents this mapping.

## For Each

This activity executes an group of activity several times and it is shown in Figure 5.16 - WPN pattern for flow activity.



**Figure 5.17 - WPN pattern for foreach activity**

In the *Check Counter* transition the *startCounterValue* and *finalCounterValue* are evaluated. The *startCounter* is grater then the *finalCounter* the patterns finishes.

## 5.4 Scopes

Scopes are a fundamental part in the compensation process. Since scopes can have fault and compensation handlers, the mappings presented here are more complex than the presented in section 5.2 and section 5.3. Although scopes have many different configurations, this section will only show a simple scope, a scope with fault handlers and a scope with fault and compensation handlers.

### 5.4.1 Simple

Represents a scope with the normal behavior.

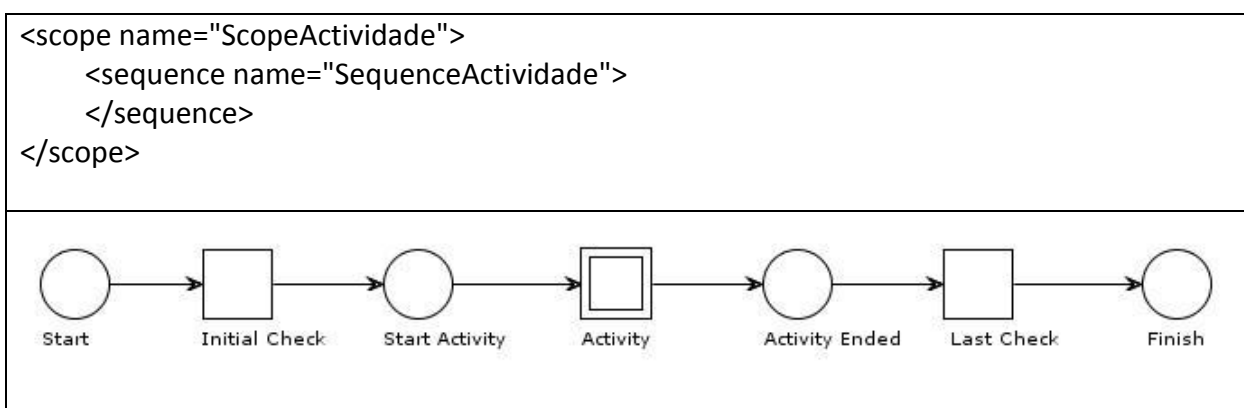


Figure 5.18 - WPN pattern for a scope

A Scope contains a sequence of activities which in WPN is abstracted, putting the activities within the sub process Activity, in order to tone down the complexity of the WPN. In this case the transitions *Initial Check* and *Last Check* are present here to change the status of the scope, used by the process to control failures. Using this definition for the scope, it is assumed that the process will handle all failures. This mapping is shown in Figure 5.18.

### 5.4.2 Fault Handlers

The Figure 5.19 presents a scope with fault handlers. The fault handlers can be defined for specific faults or describe a generic way to handle all failures. This is done using the catch tag or the catch all tag.

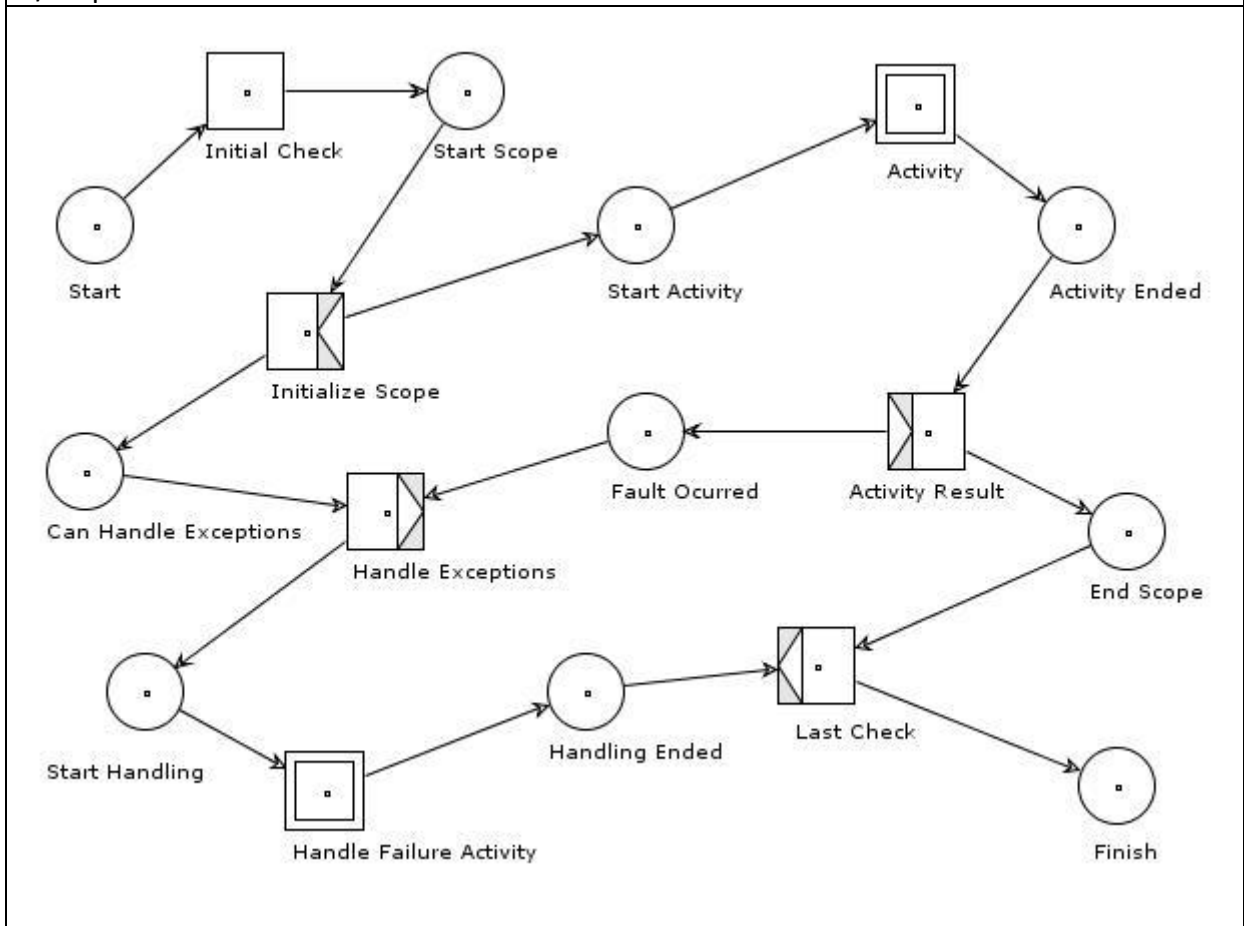


```

<scope name="ScopeActividade">
  <faultHandlers>
    <catch faultName="failure1"/>
    <catch faultName="failure2"/>

    <catch all/>
  </faultHandlers>
  <sequence name="SequenceActividade">
    </sequence>
</scope>

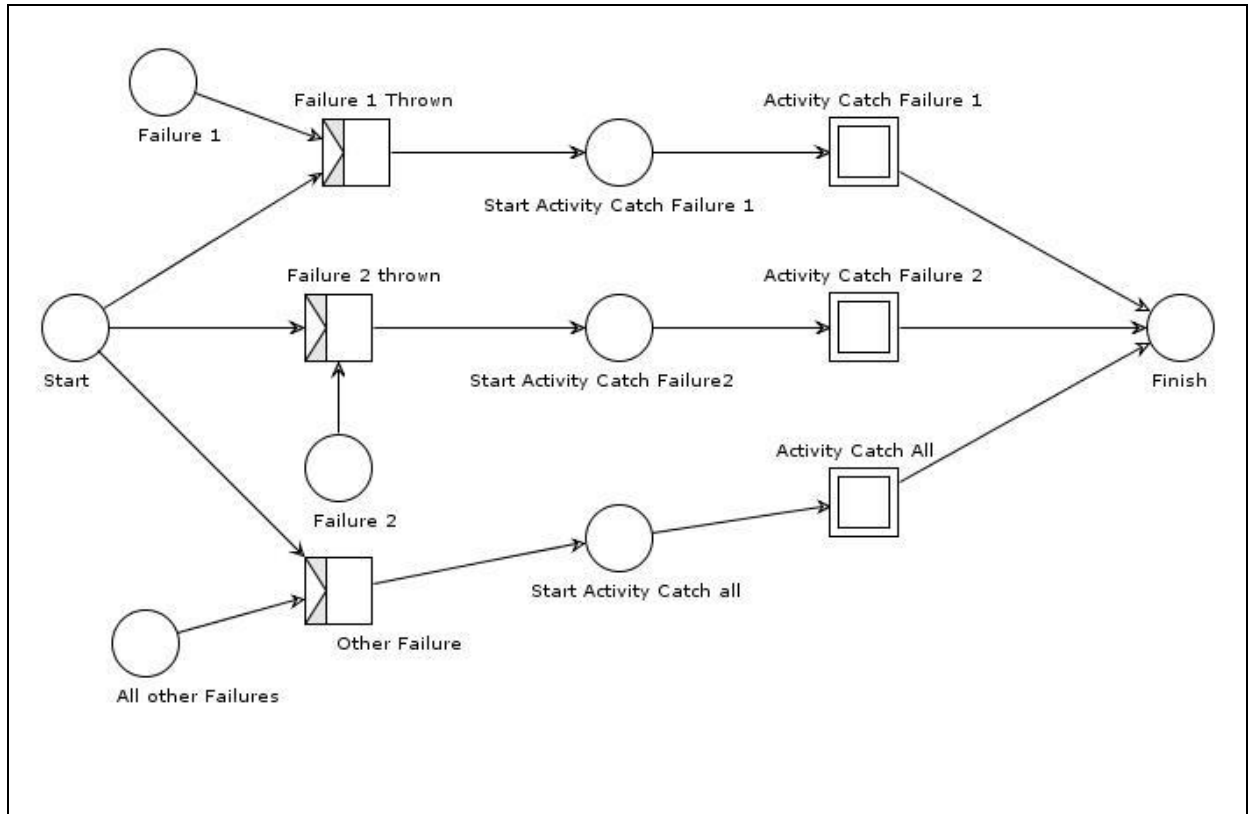
```



**Figure 5.19 - WPN pattern for a scope with fault handlers**

When a Scope has fault handlers defined, like in Figure 5.19, the activities within sub-process *Activity* can throw an exception which is treated in another sub process *Handler Failure Activity* before the end of the scope. This will affect the end result of the scope, and may start the compensation process. The transition *Initialize Scope* allows the execution of the activities of the scope, and enables the possibility to handle the failures that may occur. If a

failure occurs in the *Activity* sub process, the *Activity Result* transaction will direct the graph to handle exceptions, if not it will direct to the end of the pattern. In the first case, the *Handle Exceptions* transaction will start the failure handle activity and then go to the end of the pattern.



**Figure 5.20 - WPN pattern for the activity within the failure handlers**

The Figure 5.20 shows the definition of the *Handle Failure Activity* that is showed in Figure 5.19. It acts like the Pick pattern but the messages are substituted by the failures. If a specific failure is thrown and it has a branch associated with it, the activities in that branch will be executed. There is no limit for the number of failures to be caught. It can always be defined a branch to catch all other failures.

### 5.4.3 Compensation Handlers

The mapping for a scope with fault and compensation handlers is illustrated in Figure 5.21.

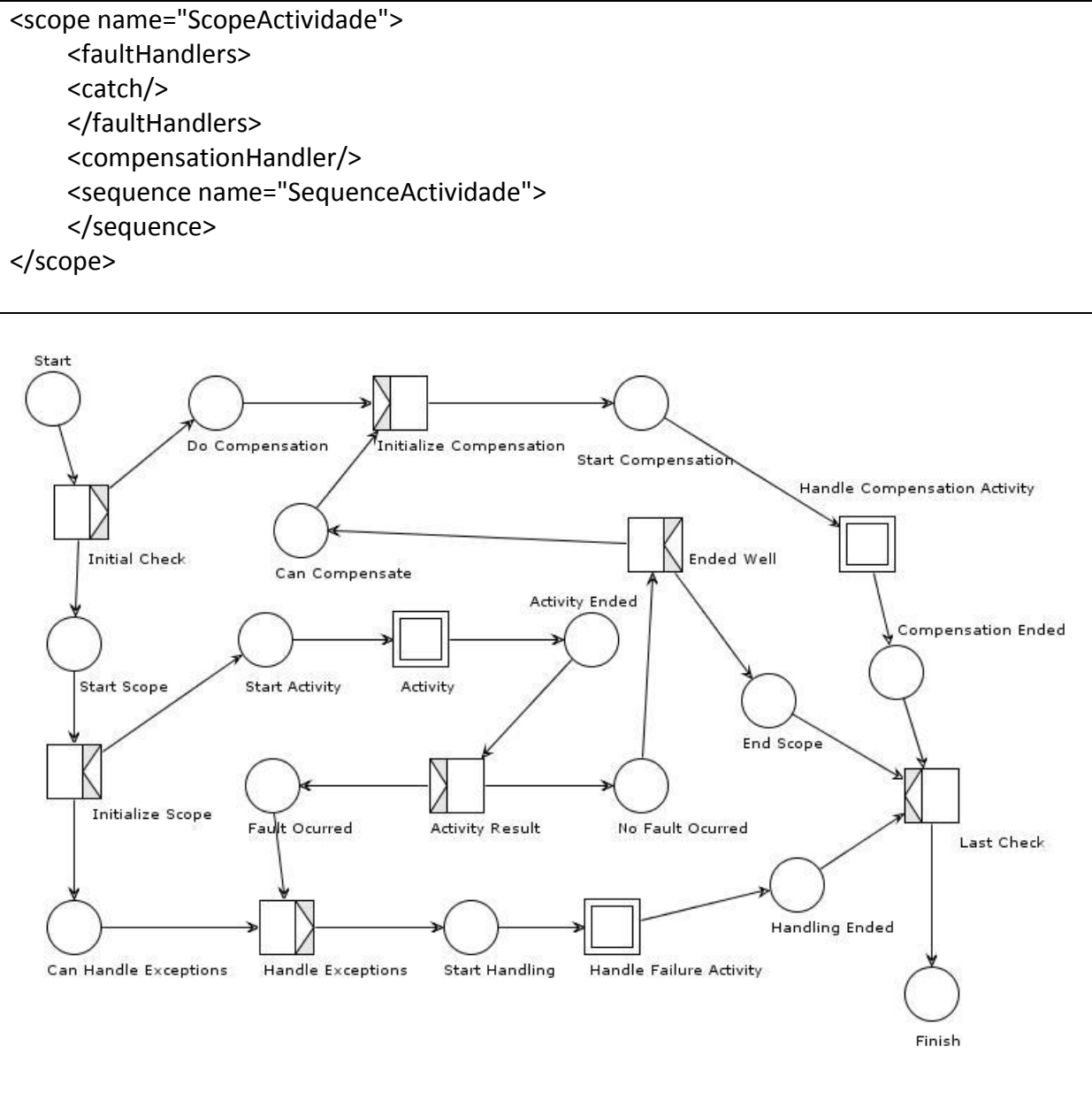


Figure 5.21 - WPN pattern for a scope with compensation and fault handlers

A scope can have both Failure Handlers and Compensation Handlers. The sub-process *Handler Compensation Activity* can only be used if this scope has already terminated before. If not, then the compensation cannot be used. The *Initial Check* transition will check if the

scope is to be initiated or to be compensated. If it is a normal execution of the scope, it must enable the possibility to handle failures. Done that, the sub-process *Activity* will execute all the underlying activities and when it ends the scope must verify if a failure has occurred. If so, it will handle the failures in the *Handle Failure Activity* and then go to the end. If the activity ended, before ending the pattern, it will enable a future compensation of the scope. The compensation is done in the *Handler Compensation Activity* and it will change the status of the scope before it ends.

## 5.5 Summary

This chapter mapped the activities of the WS-BPEL into the Workflow Petri Nets. This included simple activities and complex ones. Beside these activities, the scopes are also mapped taking in consideration the compensation and failure handlers. The mappings created form patterns that can be connected with each other. There are a few works with patterns, but are too complex to illustrate the main focus of this work or detail too much WS-BPEL activities that are not relevant. The simple approach used here will allow a better comprehension of the recovery mechanisms in WS-BPEL and provide a way to detect more easily where failures can occur.

This mapping is going to be used in the next chapter where a booking agency case study is presented.

## 6. Booking Agency Case Study

A booking agency is used as a case study for this work. The goal the booking agency is to provide a way to book a reservation in a chain of hotels. To do so, an user may access an web service running online that connects do hotels and their banks in order to make all the necessary steps to book a room at a Hotel.

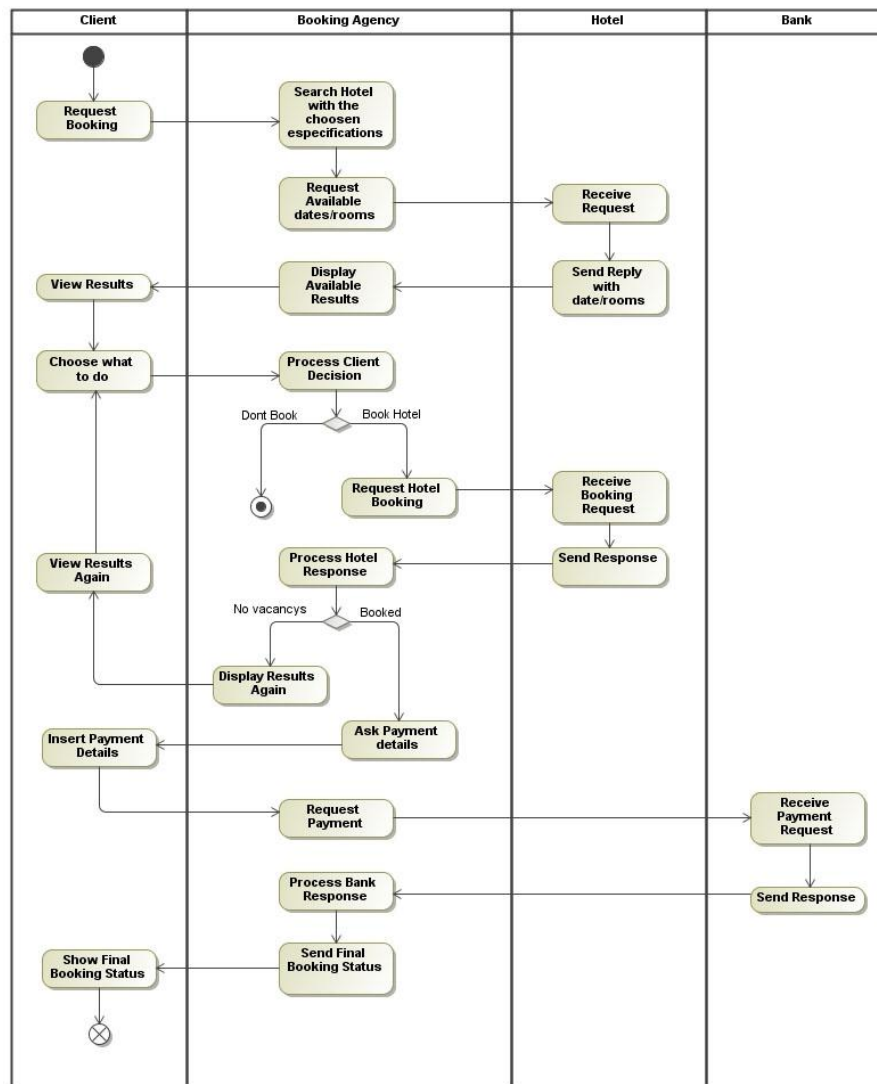


Figure 6.1 - Activity diagram for the booking process

The Figure 6.1 show the steps necessary to book an hotel, including the interactions between the partners involved in the process. First the client must specify when and what location it wants to book a room. Then the booking agency must query the hotels that match the specifications to see if they have rooms, the types of rooms and prices. A list of choices are presented to the client which will have to choose if and where he wants to book the room. If it chooses to book a room, the arrangements are made with the hotel. If the hotel is correctly booked, then the client must pay. This payment is made to the bank associated to the chain of hotels. Done all this, the process returns the booking Status.

Along with the booking process, the booking agency needs to provide a way for a costumer to cancel a booking during the duration of the whole process. Figure 6.2 shows the cancellation process where all the steps that where done in the booking process must be undone. It must cancel the reservation with the hotel and return the paid fees back to the client.

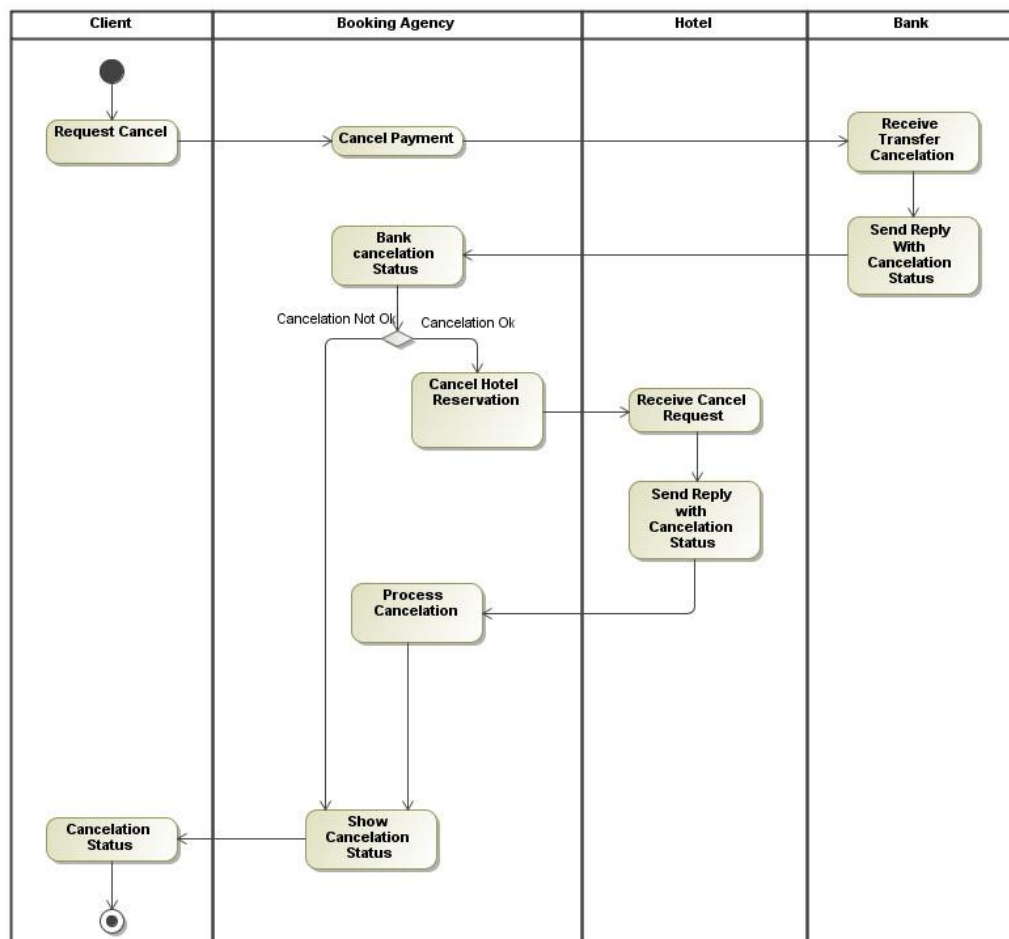


Figure 6.2 - Activity diagram to cancel a booking

## 6.1 WS-BPEL

The booking agency process was modeled using Oracle JDeveloper Studio 11.1 [11] in order to show graphically the WS-BPEL activities necessary to implement the case study. The process has four distinct fundamental steps that were divided into the implemented Scopes. These Scopes are *ScopeAvailability*, *ScopeBooking*, *ScopeTransfer* and *ScopeCancellation*. The *ScopeAvailability* has all the activities necessary to present to the client the several booking options available. The *ScopeBooking* contains the activities to book a room at the chosen hotel. The *ScopeTransfer* contains the activities involved between the client and the bank in order to pay the booking. The *ScopeCancellation* process the cancellation of a previous booking. The Figure 6.3 shows the connections between the scopes and the partners involved. The process executes the scopes in sequence and has as partners: The *Client*, *Hotel*, *Hotel2* and a *Bank*.

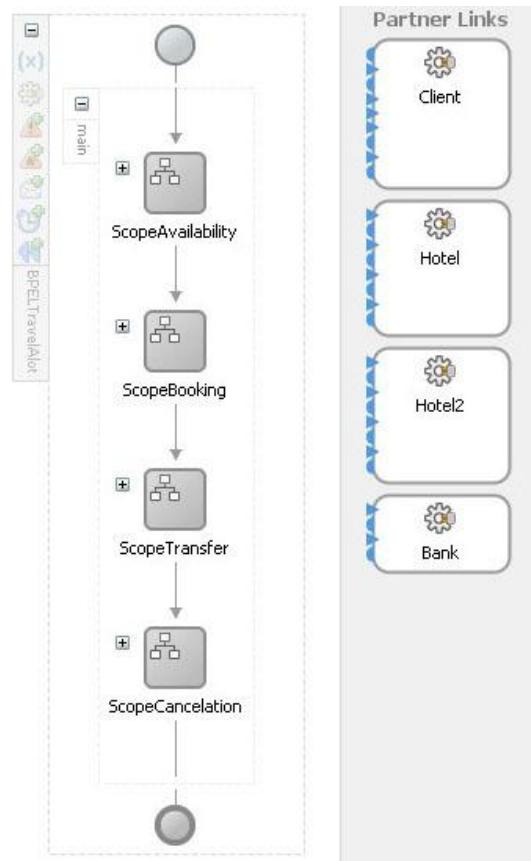


Figure 6.3 - Graphical overview of the booking agency process

### 6.1.1 Availability Activities

Before any booking, the client must first check for an available room from one of the hotels that have a partnership with the booking agency. The WS-BPEL activities involved are illustrated in Figure 6.4. The *receiveInput* receives from the client the dates and location where it wants to book a room and then the *assignDatas* copies the values to other variable that are going to be sent to the hotel partners. Since it would graphically confusing to use many hotels, our process only has two hotels, so a Flow activity will invoke on all partners there available rooms. *InvokeRoomsAvailability* invokes an operation on *Hotel* to send the rooms available on the dates chosen by the *Client*. The *InvokeRoomsAvailability2* does the same, but with *Hotel2* partner.

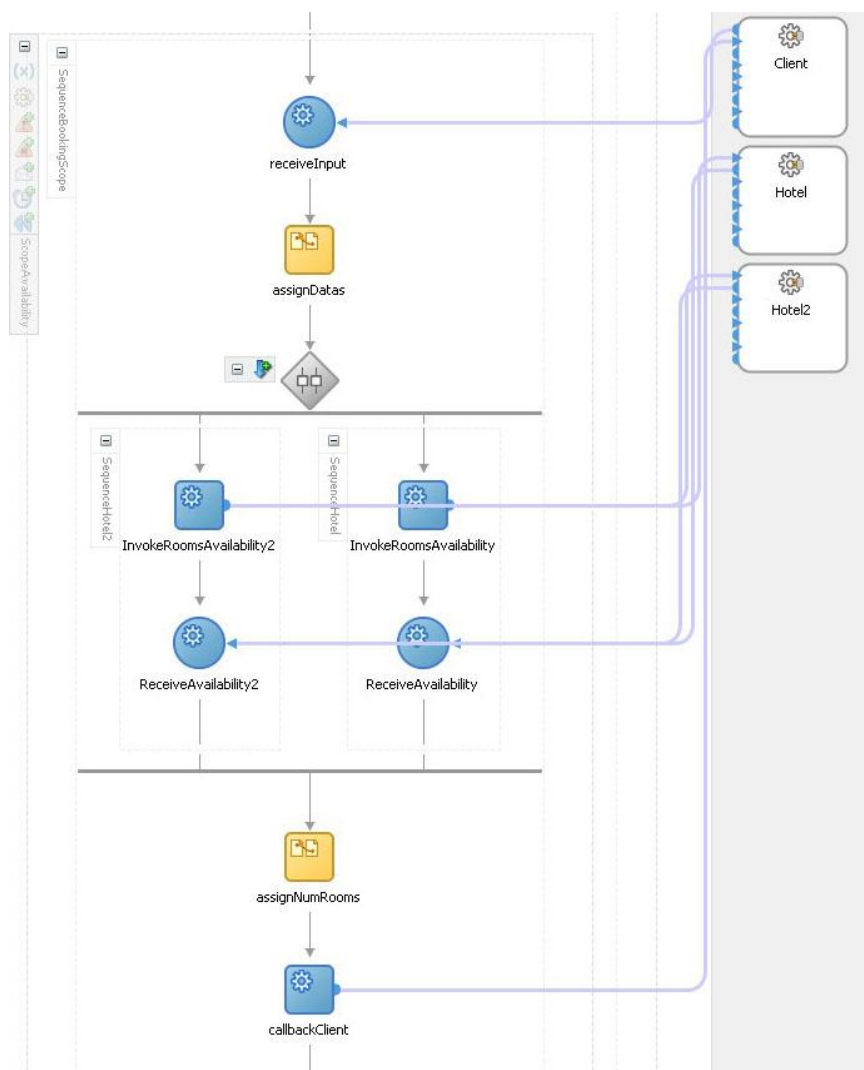


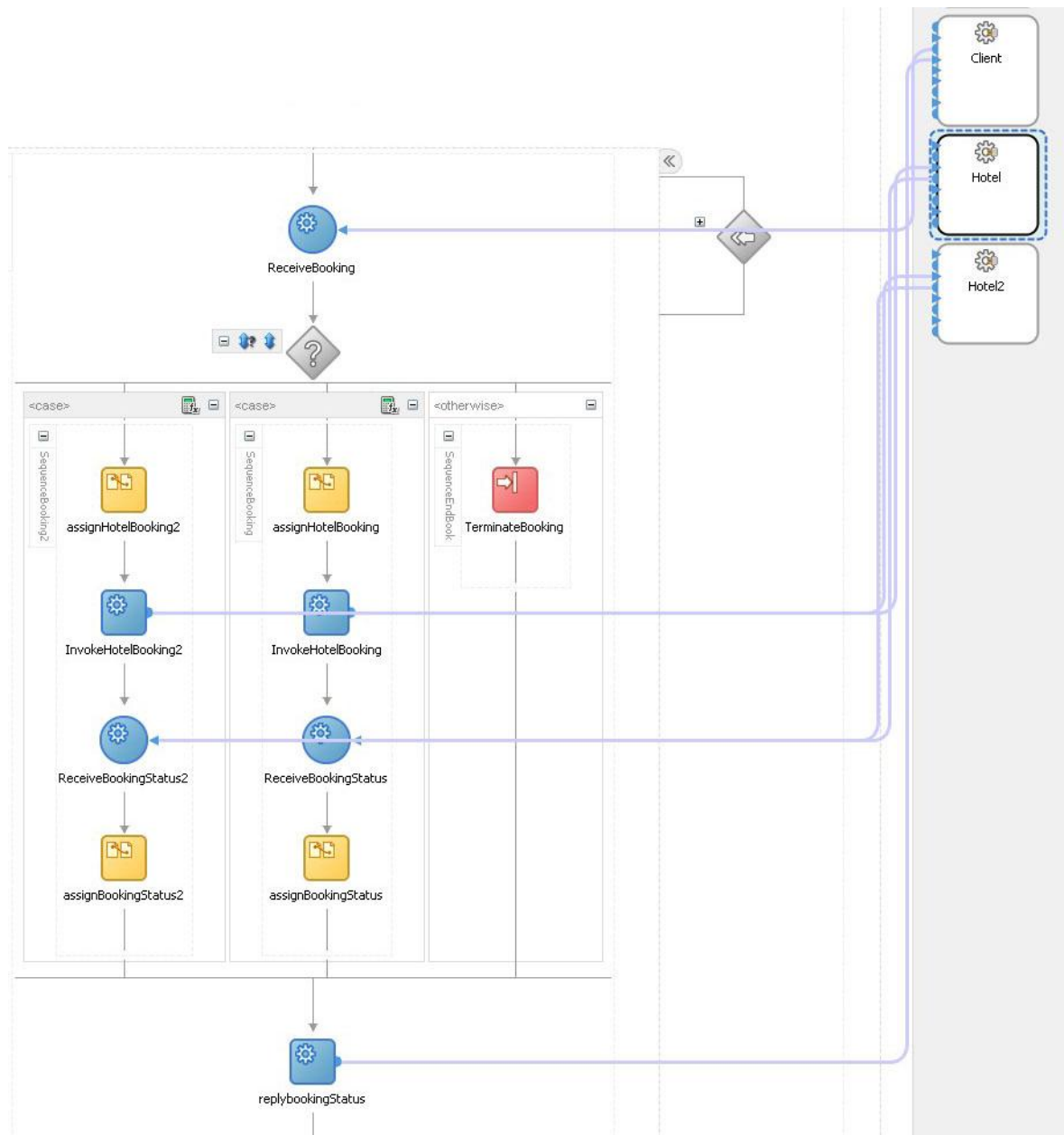
Figure 6.4 - Graphical representation of the AvailabilityScope



When the partners process the operations, they will send a response. The *ReceiveAvailability* and *ReceiveAvailability2* activities will wait for those responses, and once all arrived the Flow activity ends. The messages sent by the Hotels will be compiled into one variable in the *assignNumRooms* and then sent to the client in the *callbackClient* Activity. Once the message is sent to the client, the *ScopeAvailability* ends, and the booking may start.

### **6.1.2 Booking Activities**

The booking will start when the client has chosen the hotel, the room and the dates of the reservation. Figure 6.5 shows the WS-BPEL activities of a normal execution. The process receives a message with the decision of the client in the *ReceiveBooking* activity. After that, it processes the message and acts accordingly. The client chooses to book at one of the Hotels or it chooses to terminate the process. If it terminates, the process ends and nothing else is executed. If a client wants to refine his search of the hotels, it will start a new process from the beginning. Assuming that the client wants to book one of the hotels, the appropriate sequence of activities are executed. For each hotel they are similar because only the partner changes. Assuming the client wants to book in our partner named Hotel, the *assignHotelBooking* is executed. It copies from the client's message to another variable, the room and dates for the booking which are sent to the partner in the *InvokeHotelBooking*. Then the process will wait for a response by the Hotel if the booking was successfully in the *ReceiveBookingStatus*. Then it stores the status of the booking in another variable used later on.



**Figure 6.5 - Graphical representation of the ScopeBooking**

The activities of the *ScopeBooking* can be compensated if it is necessary. The activities that compensate this scope are displayed in Figure 6.6. First the process checks which Hotel was booked. Assuming that the booked hotel was the partner Hotel, the process will execute the *InvokeCancellationHotel1* which invokes the cancellation operation in the hotel. After invoking, the process waits for a reply from the partner in the *ReceiveCancellationHotel*. Then

it send a message to the Client with the cancelation status retrieved from the Hotel. Doing so, it ends the process, nothing more can be done.

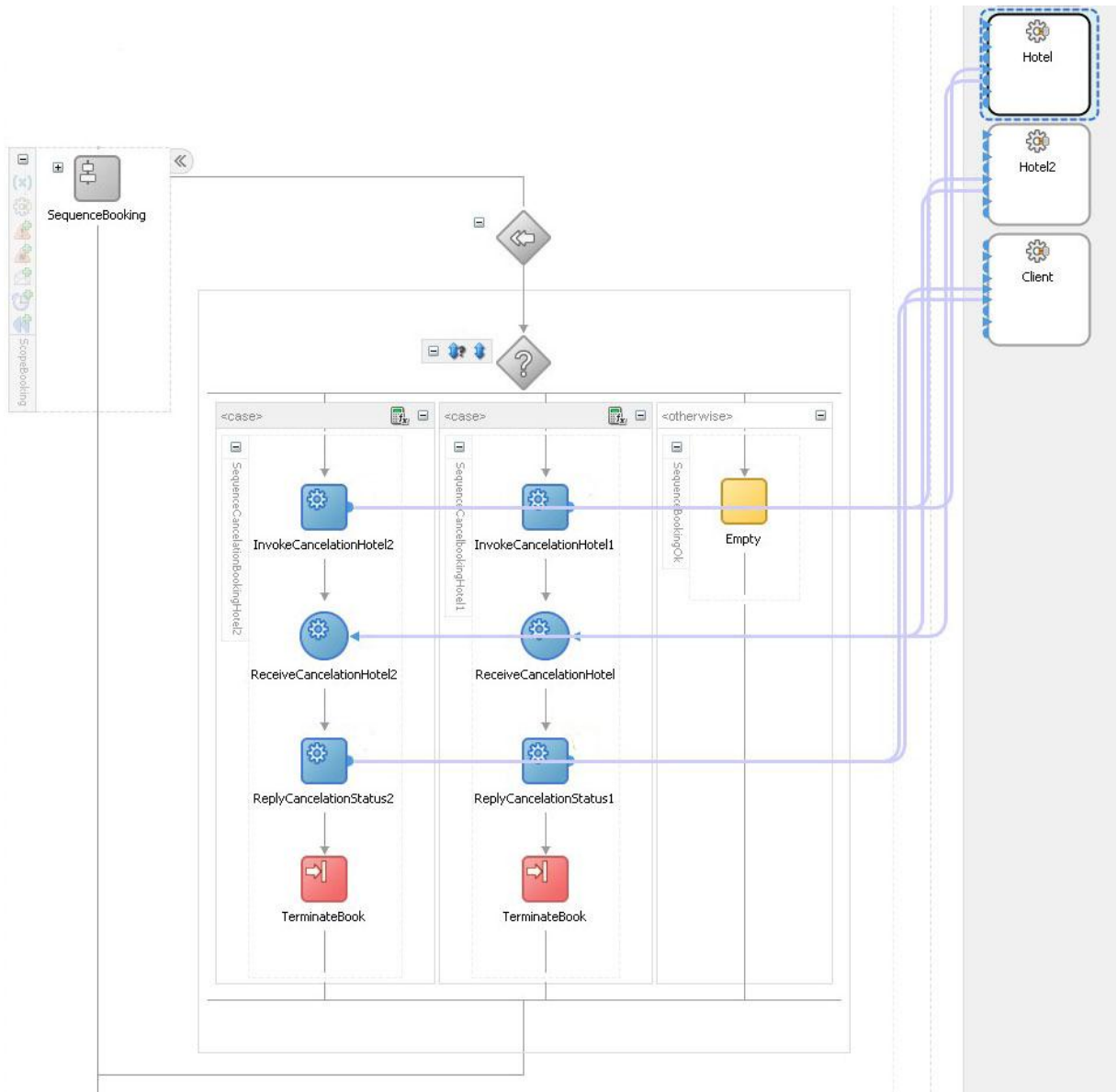
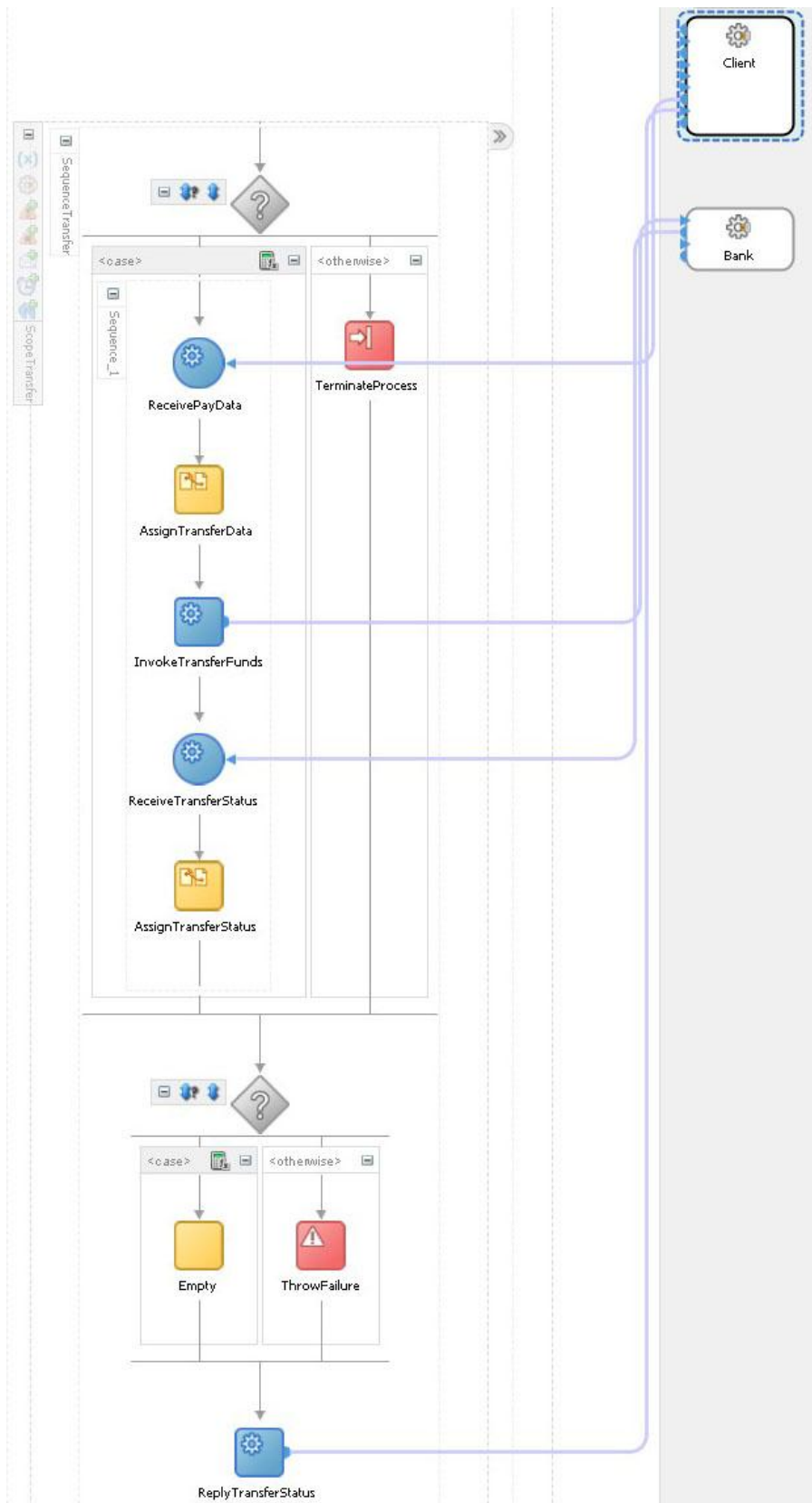


Figure 6.6 - Graphical representation of the ScopeBooking compensation

### 6.1.3 Transfer Activities

Once the booking has ended, the client must pay for the booked room. The activities that involve the Client and the Bank are contained in the *ScopeTransfer*. This scope is described in Figure 6.7.



**Figure 6.7 - Graphical representation of the ScopeTransfer**



#### 6.1.4 Cancellation Activities

The *ScopeCancellation* handles the cancellation part of the business process. In order to cancel the booking, the transfer funds must be return to the Client, and the Hotel must be notified of the cancellation. All major activities are done by the compensation handlers of the previous scopes. So this scope just waits in *ReceiveCancellationByClient* for the Client to cancel. If the client wants to cancel, the process throws an exception. This exception will be caught by the failure handlers, and it will start the hole compensation process.

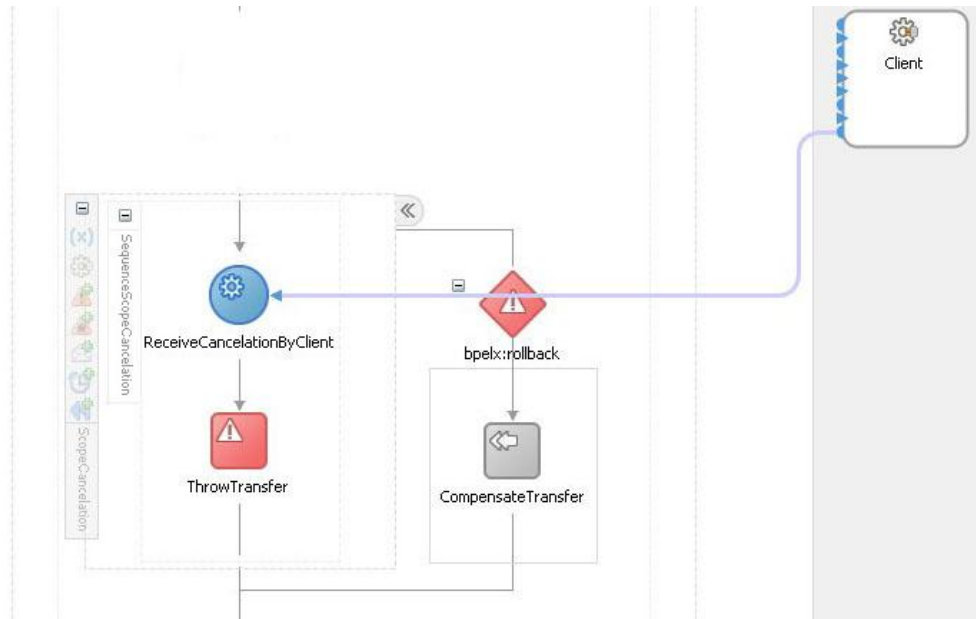


Figure 6.9 - Graphical representation of the *ScopeCancellation* and failure handlers

## 6.2 Workflow Petri Nets

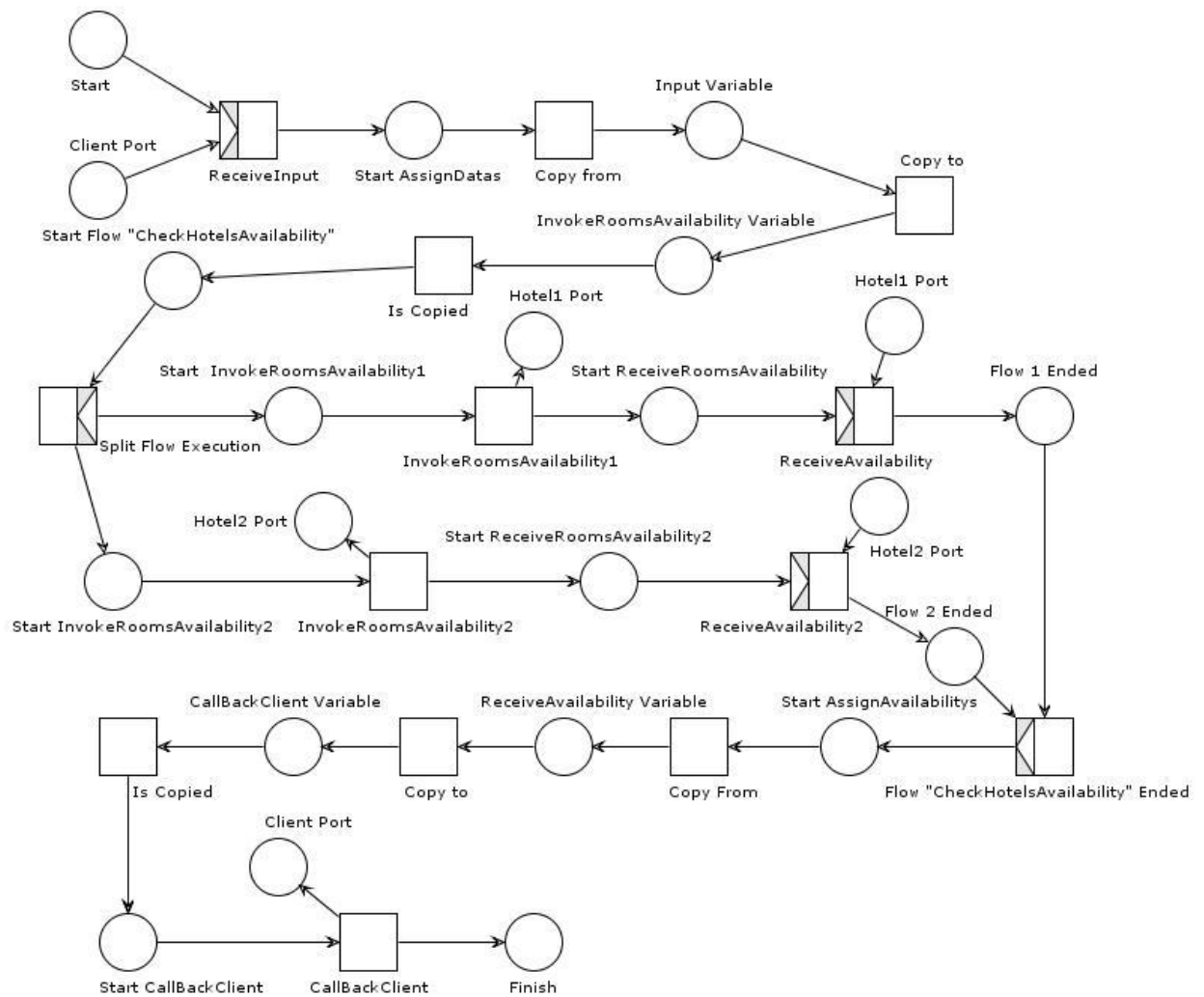
This section shows the mapping of the WS-BPEL implementation of the case study. First is presented the mapping of the scopes, failure and compensation handlers. Then a more general representation of the compensation involved in the process is explored.

### 6.2.1 Direct Mapping

The mappings that are presented here, have some particularities that have to be explained. The Workflow Petri Nets have one starting place and one finish place, but some of the mapping here will not follow that rule here, because it would lose some readability. For instance, some

mappings have a receive activity as the first operation to execute. Since it needs to be triggered by a message sent from a partner, the start place would be the port place, and the transition and-join of the receive would be a simple transition. Other option would be to add extra components which would change the receive mapping. All the places with *Port* or *Failure* that appear to be finish places are just a way to show the triggered events. The last place of the net is always the *Finish* place.

The next figures from Figure 6.10 to Figure 6.17 contain the mappings for all the scopes and the available handlers.



**Figure 6.10 - WPN mapping for the ScopeAvailability**

The *ScopeAvailability* in Figure 6.10 follows the graphical representation in Figure 6.4. The *ReceiveInput* is mapped first to a transition with the same name, associated with the port that triggers, the client. Next comes the *AssignDatas* activity and it is mapped between the place *Start AssignDatas* and the *Is copied* transition. Two concurrent flows with the same operations, but with different partners initiations. Each flow will get the available rooms from the hotels, and once all hotels have responded, the flow ends and the process continues. This flow is mapped between the place *Start Flow CheckHotelsAvailability* and the transition *Flow CheckHotelsAvailability Ended*. Next a variable is compiled with the data returned by the partners and it is sent to the *Client*, then the mapping finishes.

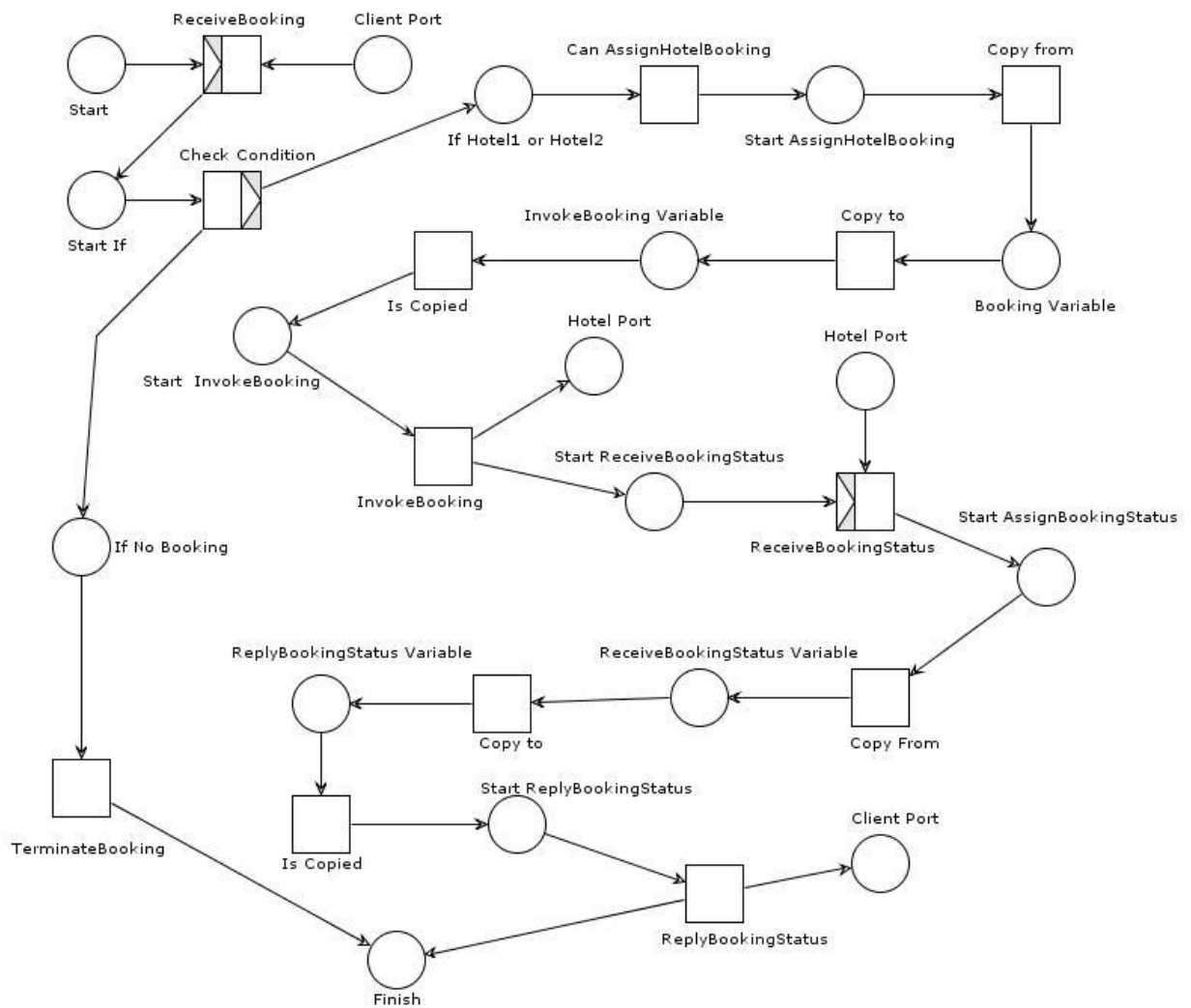
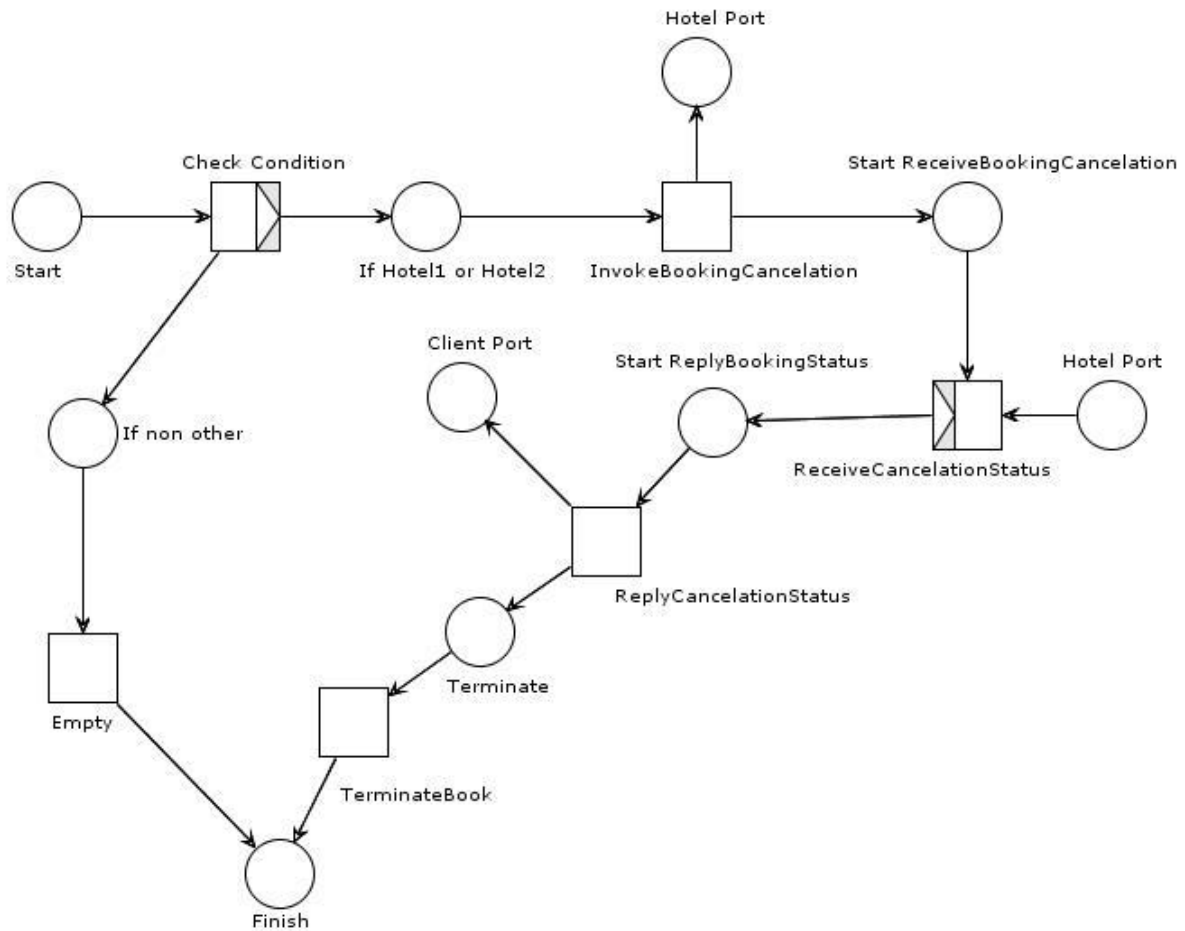


Figure 6.11 - WPN mapping for the ScopeBooking



The normal activities run within the *ScopeBooking* described in Figure 6.5 are mapped in Figure 6.11. It starts with the *ReceiveBooking* activity which is triggered by the *Client*, then the process will verify what the client has chosen and act accordingly. If the Client chooses to book a room, the path followed is the one that starts with the place *If Hotel1 or Hotel2* and ends with the transition *ReplyBookingStatus*. In between those components, variables are assigned and messages are exchanged between the process and one of the Hotel partners. If the Client does not want to book a room, the process terminates, and the flow finishes.



**Figure 6.12 - WPN mapping for the compensation handlers of ScopeBooking**

The mapping for the *ScopeBooking* compensation in Figure 6.12 is very similar with the actual mapping of the scope. First the process must check in which *Hotel* was the room booked. Once it is established the partner that is going to exchange messages with the process, in the transition *Check Condition*, the process continues by asking the cancellation of the

booking process. This is done in the *InvokeBookingCancelation* transition, then the process waits in the next transition for the feedback from the hotel. The response of the *Hotel* will then be transmitted to *Client* in the *ReplyCancelationStatus* transition. The exit activity is triggered and the process terminates in transition *TerminateBook*.

The Figure 6.11 and Figure 6.12 does not contain all the mapping for the *ScopeBooking* and its compensation. Since the booking of an *Hotel* involves the same operations and only changes the partner, those operations does not appear duplicated. Instead they are showed merged after the transition *CheckCondition* where the chosen hotel is tested, before the place *if Hotel1 or Hotel2* on both Figures.

The Figure 6.13 below contains the mapping for the *ScopeTransfer* normal execution. The graphical notation for this scope is illustrated in Figure 6.7. This mapping starts with a conditional check. The *Check Condition* transition verifies if a room was booked, if it wasn't the process terminates, if it was the process must communicate with the *Bank* in order to pay the reservation. In order for the transfer of funds occur, first the *Client* must provide the financial data necessary, this is done in the transition *ReceivePayData*. The path continues with the *AssignTransferData*, *InvokeTransferFunds*, *ReceiveTransferStatus* and *AssignTransferStatus* mappings and then another condition has to be evaluated. The status of the transfer must be checked to see if was done successfully in the transition *CheckTransferStatus*. If it was successful it proceeds and sends the status to the client and the path ends. If not, a failure is thrown and the path ends. This failure is called *rollback* and it will be tested in failure handlers defined for the scope. This is shown in Figure 6.14.

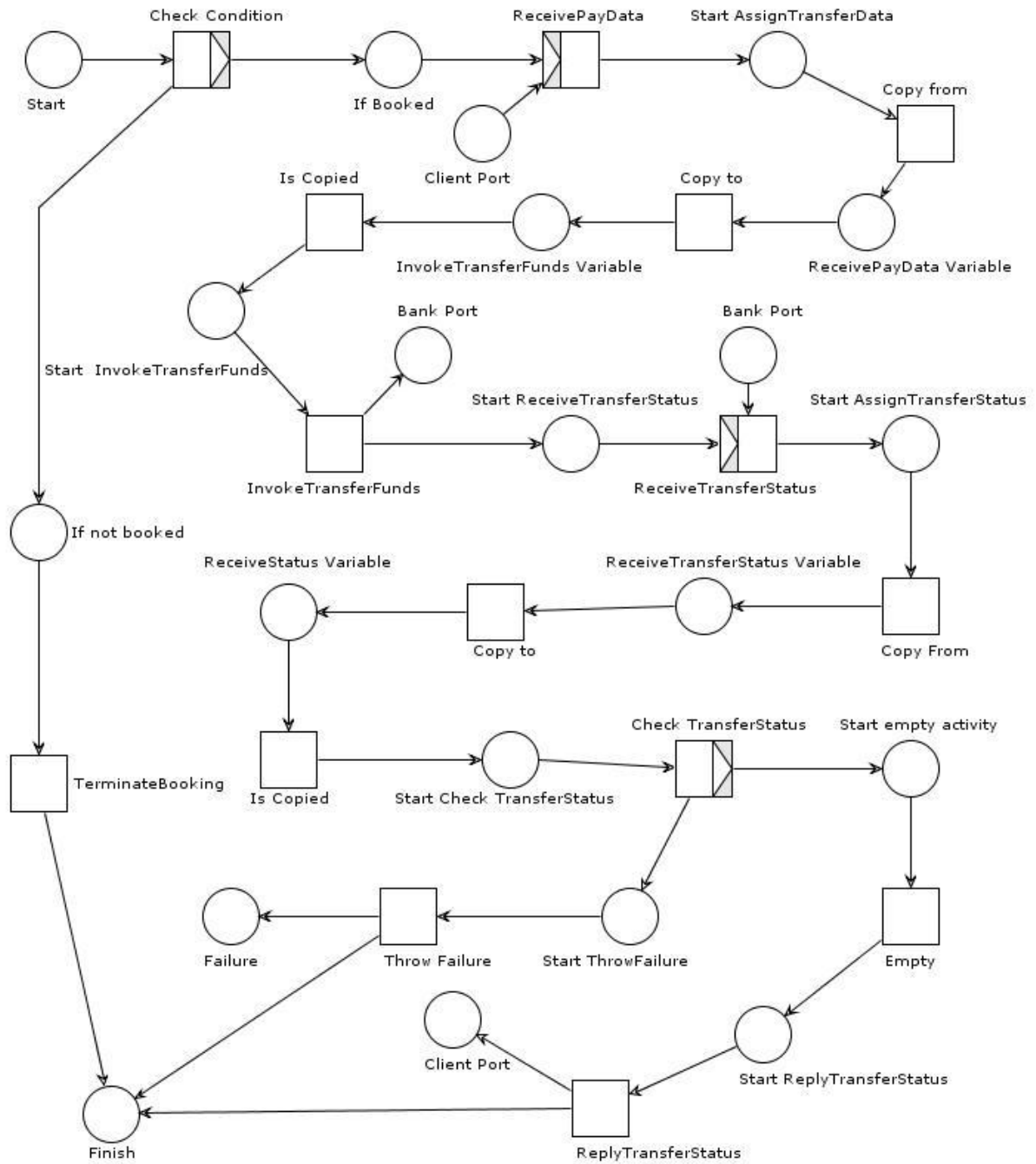
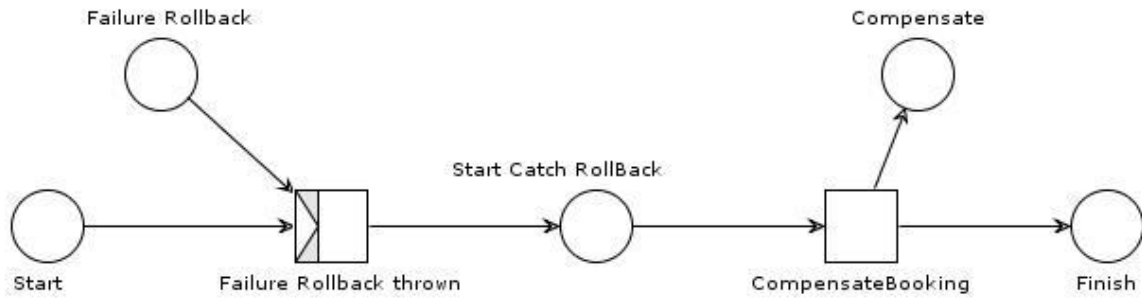


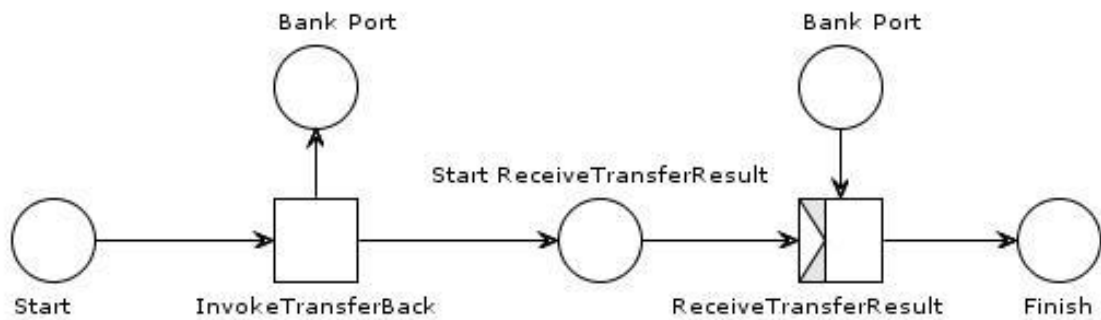
Figure 6.13 - WPN mapping for the ScopeTransfer



**Figure 6.14 - WPN mapping for the failure handler for the ScopeTransfer**

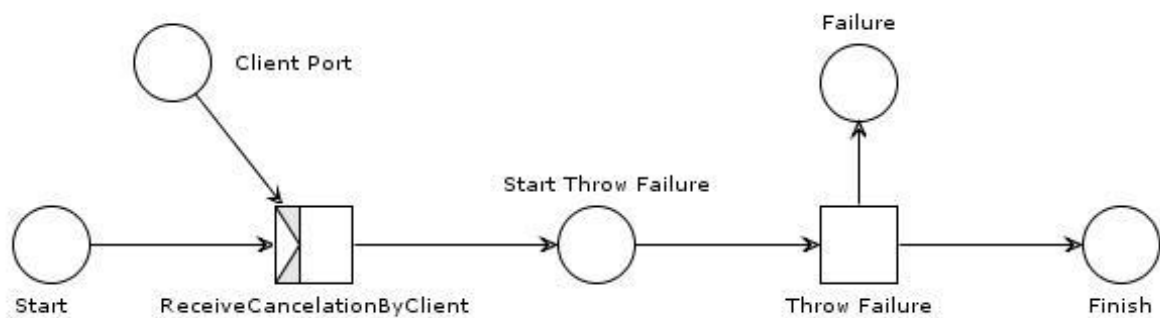
Since it was only defined one failure to be caught, the mapping of the failure handlers in Figure 6.14 is like the mapping for a sequence. If the rollback failure was thrown, then the process must call the previous compensation defined in Figure 6.12.

Figure 6.15 shows the mapping for the compensation associated with the *ScopeTransfer*. The compensation invokes an operation to transfer back the funds back to the client and the result of that invocation is return to the process.

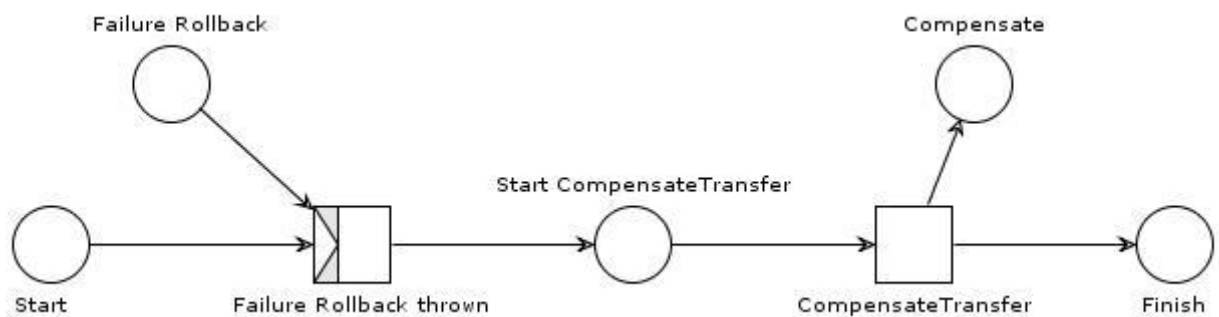


**Figure 6.15 - WPN mapping for the compensation handler for the ScopeTransfer**

The Figure 6.16 represent the mapping for the *ScopeCancellation* and Figure 6.17 for the failure handler associated. These are similar with the previous mappings presented. When a client cancels a previously booked and paid room, a failure is thrown. Again the failure is called *rollback* and it will be caught by the failure handler. Once again it is only checked for the rollback failure in the failure handler, so the mapping is like a sequence and it calls the compensation for the *ScopeTransfer*.



**Figure 6.16 - WPN mapping for the ScopeCancellation**



**Figure 6.17 - WPN mapping for the failure handler of the ScopeCancellation**

## 6.2.2 Overview Mapping

In this section the basic activities presented in the previous section are replaced here by the associated sub processes. Figure 6.18 presents a global view for the execution of the booking agency process and Figure 6.19 the details that execution with the internal work done by the Scopes.

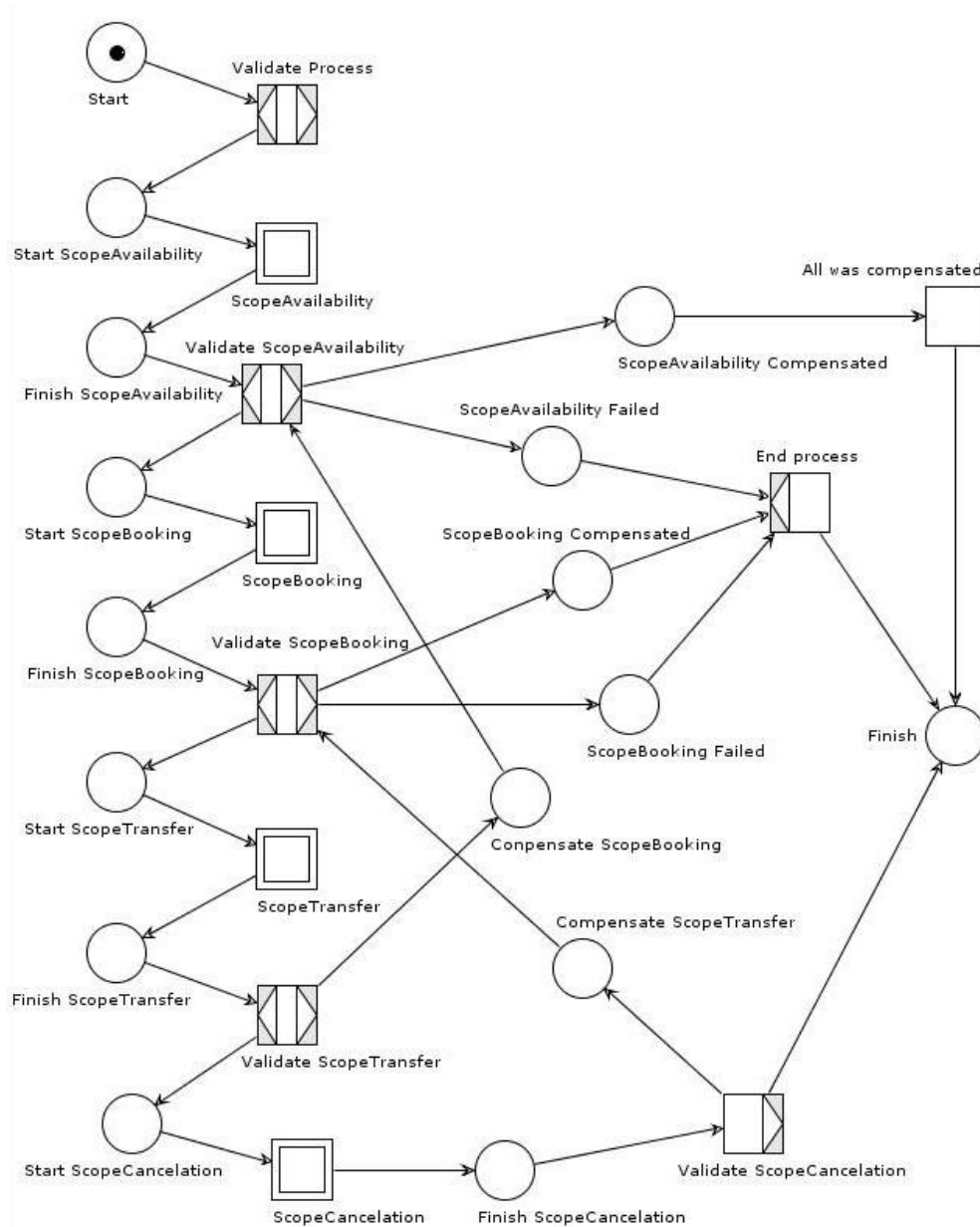


Figure 6.18 - Overview of the WPN mapping for the compensation of the booking agency process

The global view is based on the Figure 6.3 that contains a sequence of the scopes of the process. The scopes are replaced by sub processes with the same name, and the compensation is added. In the mapping chapter, the scopes are mapped as simple, with fault handlers and/or compensation handlers. They have a transition in common that is called the *Last Check*. This transition can be seen as an operation to provide the system running the process, the status of the scope and a snapshot of the variables at that time. This will be used by the process to decide what to do at the end of each Scope. This decision is made in the transitions named *Validate* before and after each scope. If a scope executed well without failure, the next scope starts its execution, if not, the compensation for the previous scope is called. If the compensation of a scope is executed, then in the end, it must execute the compensation of the previous scope like it would if the scope failed. This is done until the scopes of the process are all compensated. In this case, since there is no compensation defined for the *ScopeAvailability*, it should end all compensations after compensating *ScopeBooking*. Because of the lack of compensation on the first scope, if the second scope fails, the process will end the execution. If the first scope fails, the process also ends the execution. The Figure 6.18 shows the compensation in this case study, but in reality, if necessary the last compensation to be called would be the one in the *ScopeAvailability*. Since it is not defined, it would act as an *Empty* activity. Therefore only in case of failure of the *ScopeAvailability* the process would terminate, instead of a failure in *ScopeBooking*.

The overview of the process is detailed in Figure 6.19 following the mapping for the scopes, but introducing the notion of decision where the next step to make after the execution of each scope is decided.





Some problems arise from the mapping done here and need to be explained. The major issue is that the client most of the times would not want to cancel a reservation and in the mapping used, the process always flows through all scopes. Like it was said previously, the mapping have no notion of time, therefore the operation to cancel that can be triggered by the Client may never occur, so the process must die after a designated time frame, probably the last day of the booked room in order to provide a way for a refund. Other option could be to split the process in two, separating the cancelation part, but that would involve code all the compensation steps in the normal execution of the new process and instead of using the variables defined, go to a databank to get the values needed. Other problem that might occur is failure to communicate with a partner. Invoking an operation on a partner should be attempted several times if it fails. In this case study if a failure occurs, for instance while booking the hotel, the process ends without the Client knowing. A failure inside the compensation is not threaten. During the compensation, if a failure occurs nothing is done to prevent and may end the process without doing what it was supposed to do. In order to minimize the last problems, it should be added to the specification of the process, other scopes inside the compensation and failures handlers. Adding scopes allows the implementation of more compensations that will have more chances to treat problems, but adding also more complexity to the process. Complexity is something than can easily appear while implementing recovery mechanisms and that will take a lot of time. Nesting more scopes with failure and compensation handlers, or defining activities for every failure that can occur in some cases should be avoided because of the maintenance problems that it may provide. Since the compensation works with the snapshots of the state the scope was when it finished, the compensation rollback process may mislead the process into doing something that is undesired. If the compensation of the *ScopeTransfer* is activated, then the *Bank* should return the paid fees to the *Client*. This may not return an failure, but it may not be able to return for some unforeseen event, then the compensation will continue to cancel the reservation at the hotel. So this may leave the process to cancel the hotel room, but no refund to the client. On other hand, the refund can be done correctly, but the Hotel continued with the reservation active. If this is done over the weekend, the transfer may be pending, and the cancelation may already been done. To solve this problems, the partners should provide compensations for the cancelations process between them, which cannot be maintained by the booking agency.

### 6.3 Other Compensation Features

There are many ways to treat failures within WS-BPEL that were not used in the case study, this section will show them.

#### Invoke

A compensation handler can be defined within the *Invoke* activity. If the compensation is defined then it can be called instead of using the compensation by default.

#### Compensation Handler Instance Group

If there are several instances of the scope, usually within a construct that repeats itself, and the compensation is invoked, the compensation handlers for all child scopes instances will be called *Compensation Handler Instance Group*. If the default compensation is called, the *Compensation Handler Instance Group* will contain the compensation handler for all enclosed scopes that completed successfully, but in case of a specific compensation, it contains the installed compensation handler instances of the scope. If the compensation activities ends or a fault occurs while executing those activities is uncaught, all running instances of the scope must be terminated, and no further compensation can be made for the scope. If a scope compensated by name is within a non parallel loop activity, the invocation of the compensation is done in the reverse order of the execution. In parallel loops and event handlers, no order is specified for the scope compensation.

#### Compensation within Handlers

Compensation can be made within the Fault, Compensation and Termination Handlers ( FCT-Handlers). If a scope is defined inside one of the Handlers, then its compensation handler is only available during the execution of the enclosing handler. The main scope enclosed in a handler cannot have a compensation handler, but others nested inside can. This rule must be statically enforced because it is not reachable from anywhere within the process. This is exemplified in Figure 6.20. In this examples, the *Scope 2* within the failure handler or the compensation handler cannot have a compensation handler because it is unreachable. But the failure handler of *Scope 2* can compensate the *Scope 3*

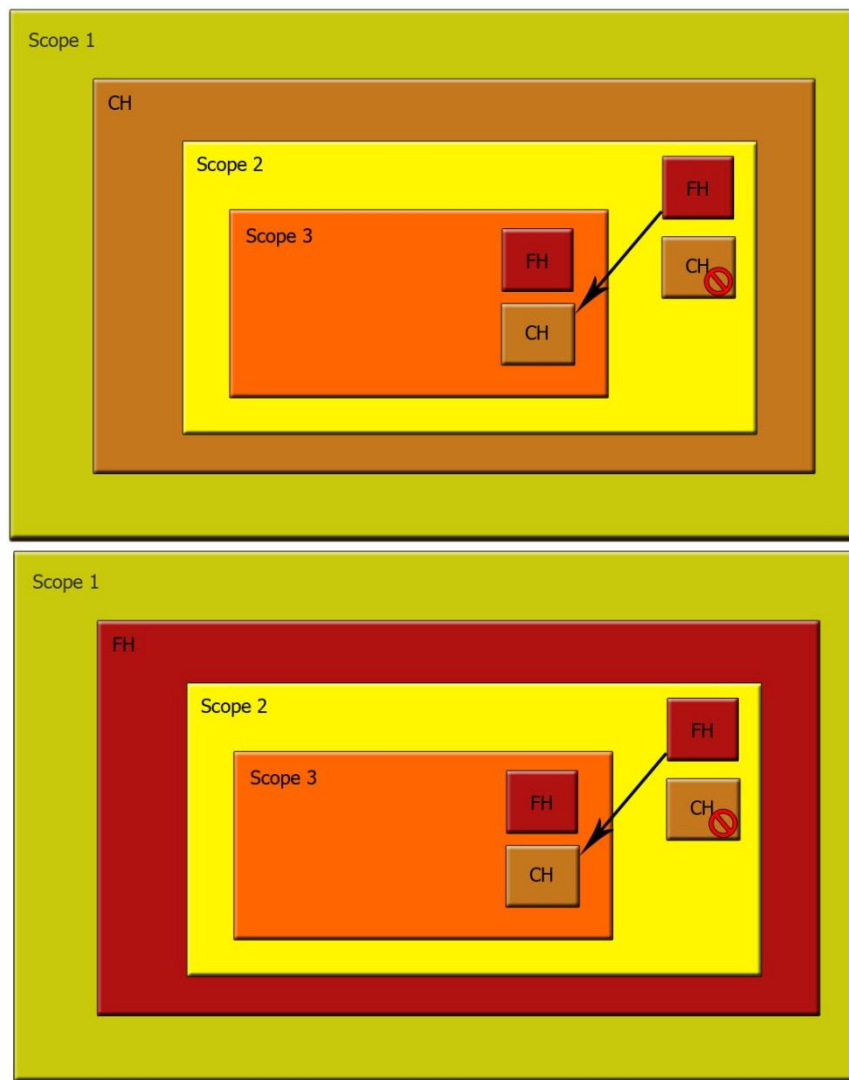


Figure 6.20 - Compensation within Handlers examples

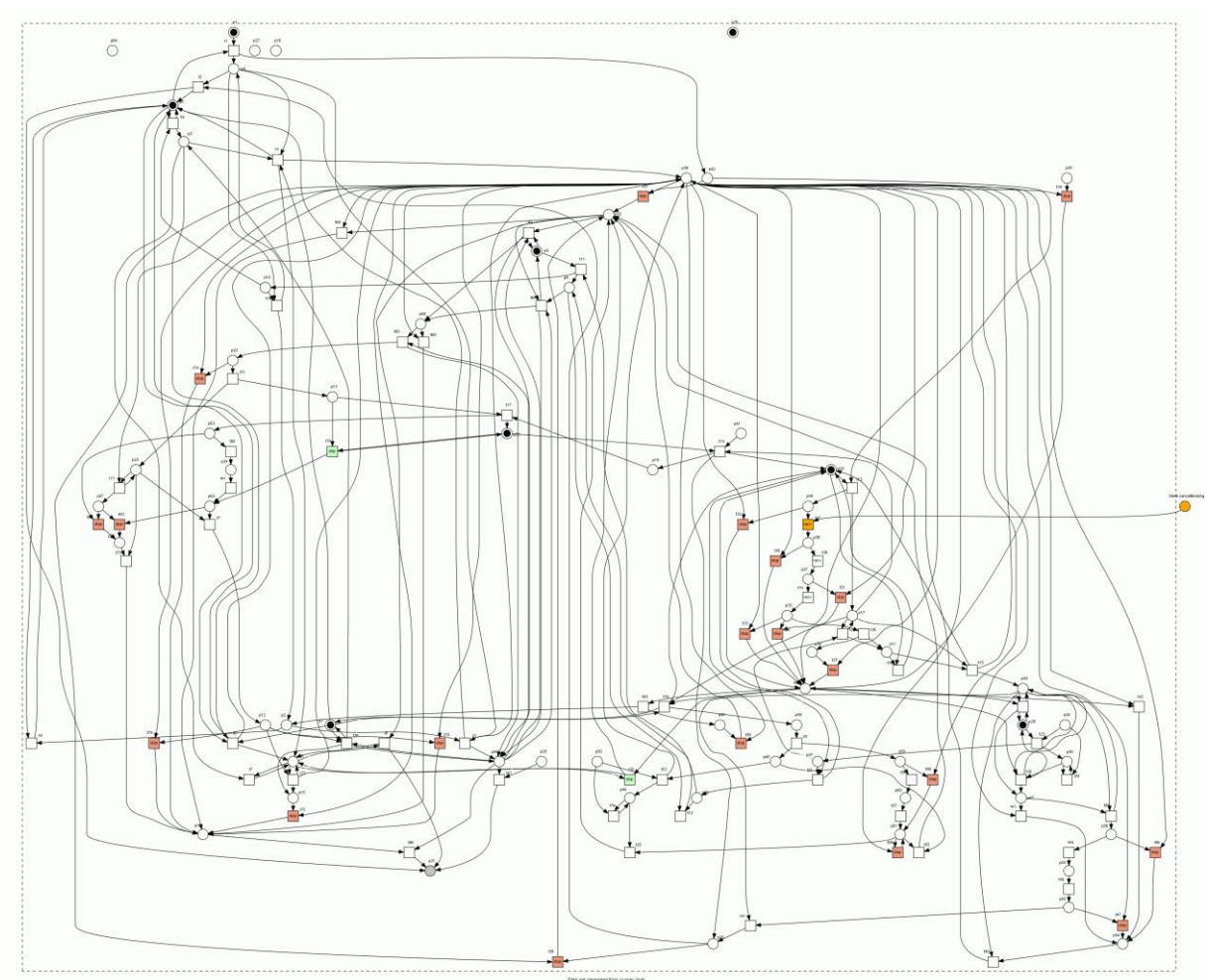
### Cyclic dependency

Scopes within this case study are threaded as isolated. When one finishes, the next one starts. So in this case it is easy to know the order of the compensations when they are needed. But when there are control links defined between activities of different scopes, these cannot form a cycle in a manner that the process can do the respected compensation because there is no way to decide which would be compensated first. The definition of the process does not allow cyclic dependency.

## 6.4 Comparing BPEL2oWFN

Since the mappings presented on this work have the purpose of showing with simplicity the recovery mechanisms presented on the business process, they will be compared with the mapping created by the BPEL2oWFN compiler presented in section 4.2.1. To do so, a portion of the case study will be mapped in both approaches, more specifically the *ScopeCancellation*.

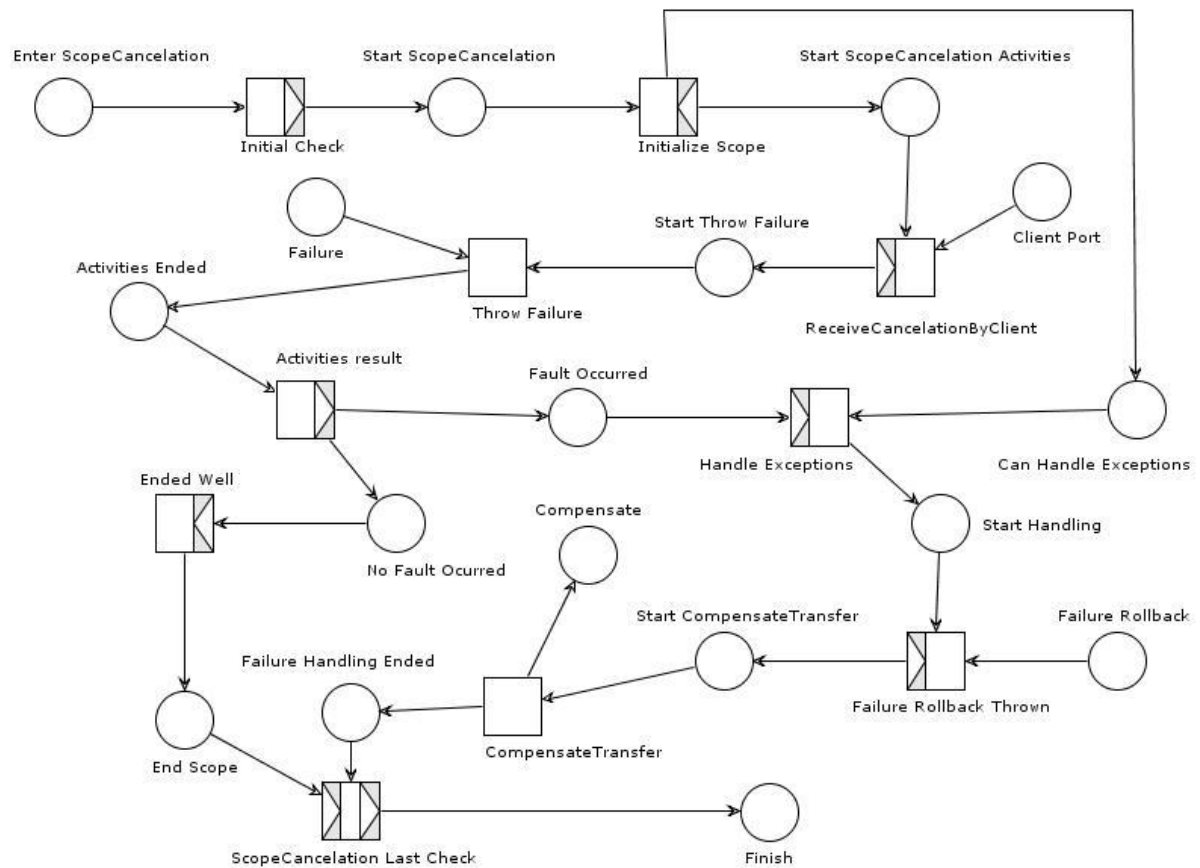
Using BPEL2oWFN compiler for the WS-BPEL file containing only the *ScopeCancellation*, created a dot file which was used to create a graphical representation (Figure 6.21) in Graphviz [29]. The compiler also showed that the WS-BPEL code was transformed into a Petri Net with 66 places and 80 transitions.



**Figure 6.21 - Graphical representation of ScopeCancellation using BPEL2oWFN and Graphviz**

The detail showed in Figure 6.21 is very difficult to comprehend. Even if the image used was in full-size, places and transitions had clear associated names, the number of arcs between the components does not allow a clear perception of the work done by the WS-BPEL.

The WPN created using the mapping of this work only has 17 places and 10 transitions. Using less detail it is more clear, to who is interpreting the Nets, the goals, activities and handlers available in the *ScopeCancellation*. The graphical representation is shown in Figure 6.22.



**Figure 6.22 - Graphical representation of ScopeCancellation using this work mapping**

Since this work focus the recovery mechanisms, and comparing both Figure 6.21 and Figure 6.22, it is easier to comprehend the concepts presented using the mapping provided instead of, for instance, the one created by BPEL2oWFN.

## 6.5 Summary

The case study presented showed how the WS-BPEL reacts to failures and how it can recover. It was created a WS-BPEL process to handle a booking agency that has the main goal to provide a way to book rooms using web services. This WS-BPEL process was illustrated graphically in order to make a visual correlation between WS-BPEL and the WPN mapping. Using the mapping it was able to show the steps of the WS-BPEL process, when it executes normally and when it fails. It was also demonstrated some shortcomings of the WS-BPEL implementation and other ways to compensate not included in the case study. The WPN mapping was compared to the one created by Lohmann in order to show how they represent the same WS-BPEL code.

## 7. Conclusion

In order to show the recovery mechanisms implemented by the WS-BPEL process, it was necessary to find the fundamentals on which it is based. Sagas, transactions in database and exception handlers in programming languages all provided basis for the recovery a treatment of failures in WS-BPEL.

There are several different formal languages with graphical notation that could be used to aid the implementation of this thesis, some of them already implemented for WS-BPEL, but focusing other aspects of the process. Workflow Petri Nets was chosen to provide a mapping for the WS-BPEL activities. This mapping helps the demonstration of the steps that are executed during the process life, including failures and compensation. The WoPeD tool used to create the mappings, provides a token ring game which allows a user to see all the paths that the process can have, and act accordantly.

The case study implemented showed that there are many ways to implement a process, and the choices made in the implementation will influence directly the mechanisms that can be used, and how they are used. Failures can be caught or thrown during the normal execution of the activities of a process. In order to treat failures, a compensation for the activities already completed must be done. This compensation is composed by the same kind of activities that were used in the normal process execution. Since logic units of work can be separated by Scopes, the compensations are associated to the Scopes to provide a rollback mechanism specific for that scope. Only when all activities of the scope have finished, it is possible to compensate. So in order to compensate a process, all the compensations of the scopes that finished earlier must be run in the inverse order of their execution. The first scope compensation will be the last to be executed.

## 7.1 Contribution and work limitations

There are a few works done around WS-BPEL involving many aspects of the business process. The major contribution of this thesis is to provide a simple way to explain the concepts behind the recovery mechanisms in WS-BPEL and how they can be implemented. Show the strong points it has and the shortcomings encountered. The mapping provided between the WS-BPEL and WPN also can be used during the implementation of a business process to path the possible scenarios that may have to be overcome once it is executed. The simplicity introduced by the mapping, will also provide a way to show everyone, not just experts in the field, how the business process works. Other works using Petri Nets are too complex and detail to much the WS-BPEL activities forming enormous patterns that make it more difficult to express the available recovery mechanisms which are the basis of this work.

Not all aspects of the WS-BPEL were mapped. *Links, Correlations, Variables* among others particularities of the activities cannot be described in the mappings provided. The tool used to create the mappings works, but it needs further development in order to become more stable and user friendly.

## 7.2 Future Work

A new tool should be developed to convert a WS-BPEL file into a Workflow Petri Net, and instead of providing places and transitions, provide patterns. Connecting the patterns with each other would in the end provide a WS-BPEL file. It must have a token game and provide the list of possible failures to test. It would be interesting if this tool could automatically add some compensation by analyzing the patterns involved.



## Bibliography

1. OASIS. *Web Services Business Process Execution Language Version 2.0* - <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>. 2007; Available from: <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>.
2. H. F. Korth, A.S., S. Sudarshan, *Database system concepts*. 5 ed. 2006: McGraw-Hill, Inc. 546.
3. Bpel2oWFN. <http://www.gnu.org/software/bpel2owfn/>.
4. Sun. <http://java.com/en/about/>.
5. Hector, G.-M. and S. Kenneth, *Sagas*. SIGMOD Rec., 1987. **16**(3): p. 249-259.
6. Moss, J.E.B., *Nested transactions and reliable distributed computing*. 1982: IEEE CS Press.
7. OASIS. *XLANG* - <http://xml.coverpages.org/xlang.html>. Cover Pages; Available from: <http://xml.coverpages.org/xlang.html>.
8. OASIS. *Web Services Flow Language (WSFL)* - <http://xml.coverpages.org/wsfl.html>.
9. W3C. *Extensible Markup Language (XML)* - <http://www.w3.org/XML/>.
10. WSDL. <http://www.w3.org/TR/wsdl>.
11. JDeveloper, O. <http://www.oracle.com/technology/products/jdev/index.html>.
12. Paul, G., et al., *Compensation is Not Enough*, in *Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*. 2003, IEEE Computer Society.
13. Microsoft. *Biztalk Server* - <http://www.microsoft.com/biztalk/en/us/default.aspx>.
14. IBM. *Websphere* - <http://www-01.ibm.com/software/websphere/>.
15. IBM. *WebSphere Process Server V6.0 Business Process Choreographer Programming Model*. Available from: <http://www-01.ibm.com/support/docview.wss?rs=2307&uid=swg27007157>.
16. Murata, T. *Petri nets: Properties, analysis and applications*. in *Proc. IEEE*. 1989.
17. Stahl, C., *A Petri Net Semantics for BPEL*. 2005, Humboldt-Universitat zu Berlin: Berlin.
18. Lohmann, N., *A Feature-Complete Petri Net Semantics for WS-BPEL 2.0 and its Compiler BPEL2oWFN*. 2007, Humboldt-Universitat zu Berlin: Berlin. p. 41.
19. Lohmann, N., et al., *Analyzing interacting WS-BPEL processes using flexible model generation*. Data Knowl. Eng., 2008. **64**(1): p. 38-54.
20. König, D., et al., *Extending the compatibility notion for abstract WS-BPEL processes*, in *Proceeding of the 17th international conference on World Wide Web*. 2008, ACM: Beijing, China. p. 785-794.
21. Ouyang, C., et al., *Formal semantics and analysis of control flow in WS-BPEL*. Sci. Comput. Program., 2007. **67**(2-3): p. 162-198.
22. David, H., *Statecharts: A visual formalism for complex systems*. Sci. Comput. Program., 1987. **8**(3): p. 231-274.
23. Baeten, J.C.M., *A brief history of process algebra*. Theor. Comput. Sci., 2005. **335**(2-3): p. 131-146.
24. Jeff Magee, J.K., et al, *Behavior analysis of Software Architectures*, in *1st Working IFIP Conference On Software Architecture*. 1999: San Antonio, USA.

25. Foster, H., *A Rigorous Approach To Engineering Web Service Compositions*. 2006, University of London: London. p. 207.
26. Aalst, W.v.d. and K.M.v. Hee, *The Application of Petri Nets to Workflow Management*. The Journal of Circuits, Systems and Computers, Vol. 8, No. 1. 1998: MIT Press. xvi, 368 p.
27. Aalst, W.M.P.v.d. and K.B. Lassen, *Translating unstructured workflow processes to readable BPEL: Theory and implementation*. Inf. Softw. Technol., 2008. **50**(3): p. 131-159.
28. Woped. [www.woped.org](http://www.woped.org).
29. Graphviz. <http://www.graphviz.org/>.