MASTER

Translation of process modeling languages

Vijverberg, W.M.

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Translation of
Process Modeling Languages

By
Wouter M. Vijverberg

Institute:	Eindhoven University of Technology
Supervisor:	Prof. dr. K.M. van Hee
	Dr. J.C.S.P. van der Woude
Period:	April 2004 – June 2006
Company:	Deloitte
Department:	Consultancy
Coach:	Dr. A. van Dijk

Eindhoven, Aug. 2006

A. Preface

The study of “Computer Science and Engineering” at the Eindhoven University of Technology (TU/e) is concluded with master thesis project. My project was in the field of Information Systems, with Architecture of Information Systems as my area of expertise. The project was partly conducted at Deloitte Consultancy.

For my project, I was looking for an assignment that could be conducted at a large company. I was interested in how it was to work in a company with multiple colleagues and how I as a Computer Scientist can be of value for a company. Therefore I was excited about the assignment at Deloitte.

Finally, I would like to thank Kees van Hee, Jaap van der Woude, Andries van Dijk, Reinier Post and Ivo Raedts for their stimulating and encouraging ideas, comments and reactions on my work. I also would like to thank my girlfriend Linda and the rest of my family and friends for the love and support during my study.

Wouter Vijverberg
Eindhoven, June 2006

B. Summary

In the industry as well as the academic world has done much research on business process modeling. There are many standards and languages developed, each with its own tools and notations. The lack of a universal process modeling language for creating business processes has resulted in a large amount of non-exchangeable models. Simulation is possible for some tools, but often not well defined and verification of behavioral properties is impossible with the current tools.

In the academic world, much research and experience is available about Petri nets. Petri nets are used for several years and many tools are available for creating and checking the process models. Petri nets are fully formalized; many techniques are available for analysis: verification of behavioral properties and simulation of performance.

In the industry many other process modeling languages besides Petri nets are used. Deloitte for instance is using EPC's, BPMN, CaseWise and Provision. None of them have formal semantics or analysis tools. Therefore we present a method to translate these models into Petri net models. A huge amount of analyzing and checking tools are available for Petri nets and after the translation we can use those tools to check the translated models.

First we look at processes models created in Provision Enterprise®. For each language construct, a translation in Petri nets is given with the same semantics. When we connect the translation of each language construct in a process model, the translation of the whole process model is obtained. The translation is implemented in a C++ application, which uses the Provision API to read the Provision model. The application creates a PNML-file, which can be opened in Yasper, the editor that has been used for modeling Petri nets.

Secondly, we look at the Business Process Modeling Notation (BPMN); a recently introduced notation developed by the Business Process Management Initiative (BPMI) to become a new standard for modeling business processes. The BPMI is a working group which is composed of several prominent companies like IBM, PeopleSoft, Lombardi Software and Popkin Software.
From the official specification an informal semantics of BPMN is derived. The informal semantics is used to give a translation of the language constructs of BPMN into Petri nets. The translation is implemented with the use of XSLT.

The translated process models can be verified and simulated with the tools available for Petri nets.

C. Contents

A. PREFACE ... **3**

B. SUMMARY ... **4**

C. CONTENTS ... **5**

D. INTRODUCTION ... **7**

D.1. ASSIGNMENT BACKGROUND ... 7
D.2. PROBLEM AREA ... 7
D.3. ASSIGNMENT DEFINITION ... 7
D.4. APPROACH ... 8
D.5. REPORT OUTLINE ... 8

1. PETRI NETS AND PNML ... **10**

1.1. OVERVIEW OF PETRI NETS ... 10
1.2. YASPER NETS ... 12
1.2.1. Summary of the building blocks ... 15
1.3. THE PNML STRUCTURE ... 16

2. PROVISION ... **20**

2.1. PROVISION ENTERPRISE ... 20
2.1.1. Business processes ... 21
2.2. LANGUAGE CONSTRUCTS ... 22
2.3. THE TRANSLATION INTO PETRI NETS ... 24
2.3.1. The translation of the links ... 24
2.3.2. The translation of the nodes ... 24
2.3.2.1. Activity ... 24
2.3.2.2. Junction ... 25
2.3.2.3. Decision point ... 26
2.3.2.4. Source ... 26
2.3.2.5. Sink ... 27
2.3.2.6. Sub processes ... 27
2.3.2.7. Reference Elements ... 27
2.4. IMPLEMENTATION ... 27

3. BPMN: INTRODUCTION ... **29**

3.1. BPMN LANGUAGE CONSTRUCTS ... 29
3.2. EXAMPLE ... 30
3.3. OVERVIEW OF THE BPMN OBJECTS ... 30
3.3.1. The nodes ... 30
3.3.1.1. Activities ... 30
3.3.1.2. Gateways ... 31
3.3.1.3. Events ... 31
3.3.2. The edges ... 32
3.3.2.1. Sequence flow ... 32
3.3.2.2. Message flow ... 32
3.3.2.3. Association flow ... 32
3.3.3. Swimlanes ... 32
3.3.3.1. Pools and Lanes ... 33
3.3.4. Artifacts ... 33
3.3.4.1. Data Object ... 33
3.3.4.2. Group ... 33
3.3.4.3. Annotation ... 33
3.4. INFORMAL SEMANTICS ... 34

4. BPMN: LANGUAGE CONSTRUCTS IN DETAIL ... **36**

4.1. ACTIVITIES 36
4.1.1. Tasks 36
4.1.2. Sub processes 36
4.1.3. Interruptible activities 37
4.2. GATEWAYS 38
4.2.1. Parallel Gateways (AND) 38
4.2.2. Exclusive Gateways (XOR) 39
4.2.2.1. Data-based exclusive gateways 39
4.2.2.2. Event based exclusive gateways 39
4.2.3. Inclusive gateways (OR) 39
4.2.4. Complex gateways 40
4.3. EVENTS 40
4.3.1. Start Events 41
4.3.2. End Events 41
4.3.3. Intermediate Events 42
4.3.3.1. Intermediate events 42
4.4. MESSAGE FLOWS 43
4.4.1.1. The communication nodes 43

5. BPMN: THE TRANSLATION AND IMPLEMENTATION 46

5.1. POSSIBLE CONNECTIONS OF AN ACTIVITY 46
5.2. THE TRANSLATION OF ACTIVITIES 47
5.2.1. Simple tasks and simple sub processes 47
5.2.2. Interruptible activities 48
5.2.3. Internal Markers 50
5.2.4. Add an extra level of hierarchy 51
5.2.5. Compensation Activity 51
5.2.6. Standard loop marker 52
5.2.7. Multi-Instance loop marker 52
5.2.8. Ad hoc 54
5.3. TRANSLATION OF GATEWAYS 56
5.3.1. Exclusive gateway (Data Based) 56
5.3.2. Exclusive gateway (Event Based) 56
5.3.3. Parallel gateway 57
5.3.4. Inclusive gateway 57
5.3.5. Complex gateway 57
5.4. THE TRANSLATION OF EVENTS 57
5.4.1. Events in normal flow 57
5.4.1.1. Translation of start events 57
5.4.2. Intermediate events 59
5.4.3. Translation of end events 61
5.5. CORRECTNESS OF THE TRANSLATION 65
5.6. IMPLEMENTATION 65
5.6.1. An extension for verification 69

6. CONCLUSION 70

7. RECOMMENDATIONS 70

D. Introduction

D.1. Assignment background

The study of “Computer Science and Engineering” at the Eindhoven University of Technology (TU/e) is concluded with master thesis project. My project was in the field of Information Systems, with Architecture of Information Systems as my area of expertise. The project was partly conducted at Deloitte Consultancy.

D.2. Problem Area

In the industry as well as the academic world has done much research on improving business processes. There are many standards and languages developed, each with its own notations. In [Figure 1] some commonly used standards for each domain are shown:

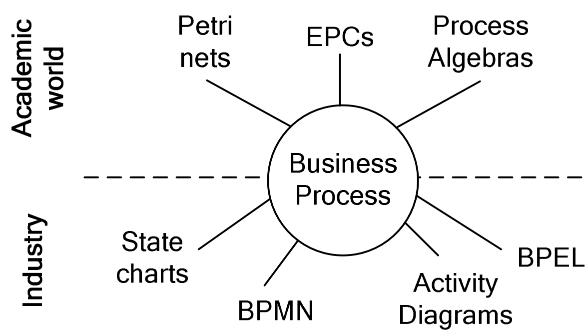


Figure 1: Business Process Domains

The conversion of a process model into another standard is often impossible and tools to simulate analyze or verify the models are often not available.

D.3. Assignment definition

The problem mentioned in the preceding section leads to the following problem definition:

The lack of a universal process modeling language for creating business processes has resulted in a large amount of non-exchangeable models. Simulation is possible for some tools, but often not well defined and verification of behavioral properties is impossible with the current tools.

To overcome the lack of verification and simulation tools for BPMN models and Provision models, a method is presented for to convert those models into Petri net models. A huge amount of analyzing and checking tools are available for Petri nets and after the conversion we can use those tools to check the converted models. This leads to the first research goal:

Find a way to translate process models created in ProVision into Petri nets, by defining the requirements for a ProVision process to be translated correctly into Petri nets and create software that implements the translation.

The second and biggest part of the project was to translate BPMN into Petri nets. BPMN has been developed to become the standard for modeling business processes. This leads to the second research goal:

Study the newly introduced Business Process Modeling Notation, design a translation of BPMN models into Petri nets, and construct software for the translation.

D.4. Approach

To meet these research goals, the research has been divided into the following parts: Petri nets and PNML, Provision: language constructs, the translation and implementation, BPMN: introduction, BPMN: language constructs in detail, BPMN: the translation and implementation.

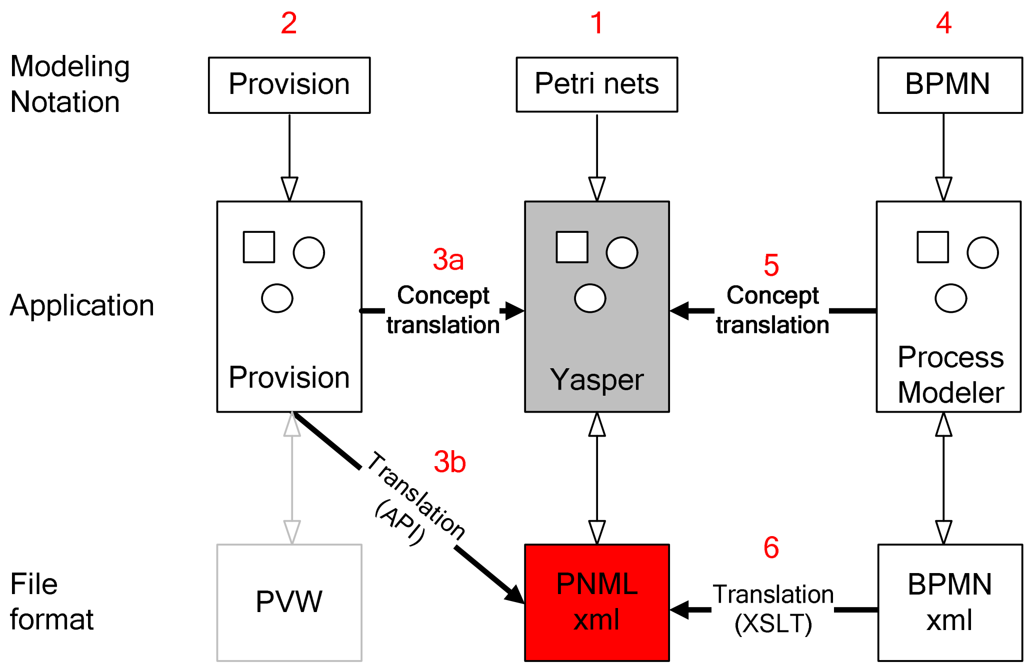


Figure 2: Research overview

In [Figure 2] an overview of the research is depicted. In the first row the modeling notation is shown. The modeling notations are implemented by applications in the second row. Within this project Yasper is used for modeling Petri nets, ProVision is used for modeling its own proprietary notation and The Deloitte Process Modeler, abbreviated as "Process Modeler", is used for creating processes with the BPMN standard. The third row shows the file format in which the process models are stored. Yasper uses the PNML format for storing the process models, which is based on xml. The Process Modeler also uses xml to save the process models. ProVision uses its own file format to store the process models, but this is of lesser importance to the rest of the research.

D.5. Report outline

1. Petri nets and PNML

For both research goals we will translate a process modeling notation to Petri nets. An introduction into Petri nets is given to understand the meaning of the language elements. Yasper uses the PNML standard to store the processes; the file format will be shown.

2. Provision

We start with an overview of the language constructs and the connections between them. A general way to translate a ProVision process into Petri nets is presented: divide the process into smaller pieces, called patterns. A translation for each pattern will be shown. With the translation of each pattern the translation of the whole process can be constructed. In the second part of this chapter, the implementation is shown.

3. BPMN: introduction
An introduction of the language constructs of BPMN is given.

4. BPMN: Language constructs in detail
An informal semantics of BPMN is shown and the syntax of the language constructs is given. Each language construct is discussed in more detail.

5. Translation of BPMN into Petri nets
For the translation of a BPMN process, the same steps as in the other translation (Provision) are followed: divide the process into patterns and give a translation of each pattern. Because Yasper as well as the Process Modeler use an on XML-based storage format, the conversion uses XSLT to convert one model into the other.

1. Petri nets and PNML

The Petri net formalism is a formal and graphical language which is appropriate for modeling systems with concurrency and resource sharing. Petri nets have been under development since the beginning of the sixties. It was the first time a general theory for discrete parallel systems was formulated. The language is a generalization of the automata theory so that the concept of concurrently occurring events can be expressed.
With Petri nets, a process can be described graphically in an easy way. Besides the fact that Petri nets are graphical, they have a strong mathematical basis and are entirely formalized. Due to this formalization, it is often possible to prove properties of the process.
Since their introduction, Petri nets have been extended in several ways to tackle more complex situations in an easily accessible way. The model developed by Carl Adam Petri [C.A. Petri, 1962] has been known as classical Petri nets. Colored Petri nets as well as hierarchical Petri nets are examples of extensions. There are many Petri net tools, e.g. ExSpect, CPN tools, Renew and Ami. However, we will use a different tool called Yasper [Yasper], developed by Deloitte and the Eindhoven University of Technology [TU/e]. With Yasper the modeler can specify and execute Petri nets in a convenient way.

1.1. Overview of Petri nets

In this document we will assume a basic knowledge about Petri nets of the reader. For those who are not familiar with them, we suggest to look at one of the following references about Petri nets, e.g. [J.L. Peterson, 1981] and [W. Reisig, 1985].

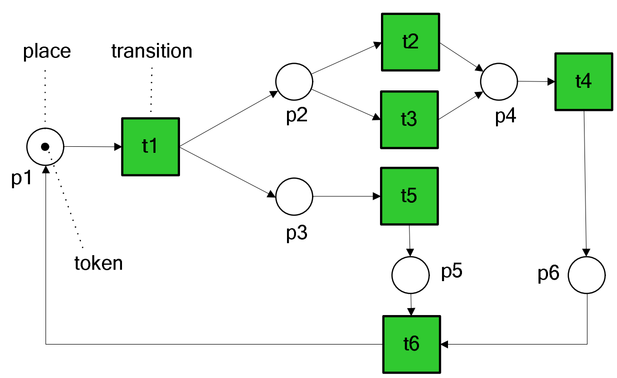
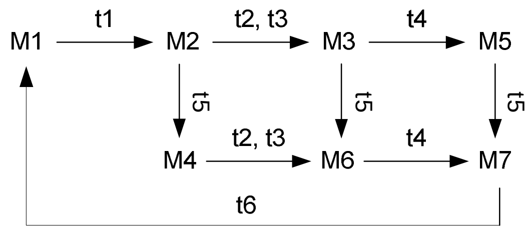


Figure **3: A classical Petri net**

Syntax
A Petri net is a bipartite directed graph. The edges are the directed arcs; the nodes are the transitions or the places. A transition is an elementary step in a process; it represents an action. A place is used to represent a state in a process. Places can contain tokens. The token distribution over the places is the state of the process, also called a marking. The initial marking is the marking the process will start with. A Petri net system is a Petri net with an initial marking.

Semantics
The semantics of a Petri net is described by a transition system. It describes the transitions between markings. The firing of a transition is the consumption of input tokens to produce output tokens. Firing of a transition will change the marking of the process. For each Petri net system a reachability graph can be constructed. It describes all markings that are reachable from the initial marking.



Markings

M1 = {p1}
M2 = {p2, p3}
M3 = {p3, p4}
M4 = {p2, p5}
M5 = {p3, p6}
M6 = {p4, p5}
M7 = {p5, p6}

Figure 4: A reachability graph

In [Figure 4] the reachability graph of the Petri net in [Figure 3] is shown. The process has seven markings: M1 … M7. The process in [Figure 3] has marking M1. Only the transition *t1* is enabled for firing. When it fires, the state of the process is changed to M2. In marking M2 there are three transitions enabled for firing: *t2*, *t3* and *t5*. The firing of *t2* or *t3* will result in marking M3, while the firing of transition *t5* will result in marking M4, etc.

Different classes of Petri nets

Petri nets can be divided into several (not necessarily disjoint) classes: e.g. state machines, workflow nets, free choice nets, marked graphs. We are in particular interested in <u>workflow nets</u>. A workflow net is a Petri net with a clear starting and ending point; the start place and the end place. Every other place or transition is on a path between the start place and the end place. For workflow nets, the <u>initial marking</u> is a marking where there is one token in the initial place of the process. The <u>final marking</u> is a marking with a token in the final place of the process. A strong correctness criterion of workflow nets is the <u>soundness</u> property:

> **Def.** [property] Soundness
> A process is sound when for each marking that can be reached from the initial marking, the final marking is reachable.

When a workflow net does not meet the soundness property, the process often contains an error, or it is an incorrect model of the process. Other structural properties of Petri nets such as liveness, boundedness or deadlock can be found in [J.L. Peterson, 1981].

Extensions of classical Petri nets

There are many extensions of classical Petri nets. Some add properties that cannot be modeled in the original Petri net (e.g. timed Petri nets). Some extensions are mentioned below:

<u>Token color</u>

In a classical Petri net all tokens are indistinguishable. In a colored Petri net, every token has a value, often referred to as "*color*". A transition describes the relation between the values of the input tokens and the values of the output tokens. Within a transition, preconditions on the values can be set: if not satisfied, the transition will not fire. For more information about colored Petri nets see [K. Jensen, 1981].

<u>Hierarchical Petri nets</u>

Process models for real systems have the tendency to become large and complex. Hierarchical structuring is used as an abstraction mechanism to make constructing, reviewing and modifying of the model easier. The hierarchical construct is called a subnet. A subnet is a Petri net by itself and can contain places, transitions and other subnets. The term "high-level Petri net" is often used for formalisms that extend the classical Petri net formalism with hierarchy. For more information about hierarchical Petri nets see [R. Valette, 1981]

<u>Time</u>

There are several ways to introduce <u>time</u> into Petri nets. Time can be added to places, transitions or tokens. When, for example, time is added to tokens, each token has a timestamp. When a transition is enabled for firing, it fires instantaneous. A transition is enabled for firing when the timestamps of all input tokens is less or equal than the current global time. The output tokens receive a new timestamp that is equal or greater than the time of firing of the transition. The transition that fires creates new timestamps for the output tokens and determines the delay. [K.M. Van Hee, 1994]

Inhibitor and reset arcs
Another extension introduces new types of arcs: the *inhibitor arc* and the *reset arc*. When an input place of a transition is connected with an *inhibitor* arc, the transition is blocked when there is a token in that input place.

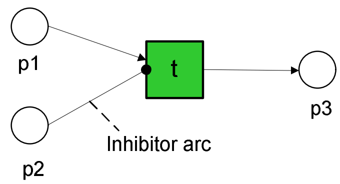


Figure 5: Inhibitor arc

In [Figure 5] transition *t* is enabled for firing when there is a token in *p1* and no token in *p2*. The transition is blocked when place *p2* contains a token.

When an input place of a transition is connected with a *reset* arc, the transition will remove all the tokens in that place when it fires.

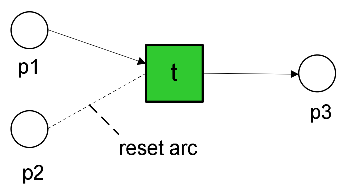


Figure 6: Reset arc

In [Figure 6] transition *t* is enabled for firing when there is a token in place *p1*. When transition *t* fires, a token is consumed from place *p1*, all the tokens in place *p2* are removed and a token is produced in *p3*, atomically.

1.2. Yasper nets

Within this thesis Yasper has been used to model Petri nets. Yasper is a software tool designed to work with workflow nets. It also supports some extensions: a limited application of token colour, inhibitor arcs, reset arcs, time and hierarchy. Yasper has a simulation function, where the processes can be executed, automatically or manually.

Case identity as color
A limited application of token colour is implemented to support concurrency among cases in the process. Each token in the initial place represents a case in the process (e.g. a single product in production). There are two types of tokens: case tokens and normal tokens. Each case token has an id, the case id, which shows to which case it belongs, but the normal tokens don't have an identity. Tokens with the same *case id* belong to the same case. To support the different token types, there are case-sensitive places, which can only hold *case* tokens. Any other place can only hold tokens without case id, the normal tokens. Case tokens or normal tokens are generated at a certain rate at the starting point of the process, the emitor. An emitor creates tokens on all its output places at once. A collector, the ending point of the process, consumes the tokens it receives like a transition. A case token must travel on a case sensitive path from the emitor to the collector. When all the tokens of a specific case are collected by a collector, the case is marked as complete.

Example: (yellow places are case sensitive)

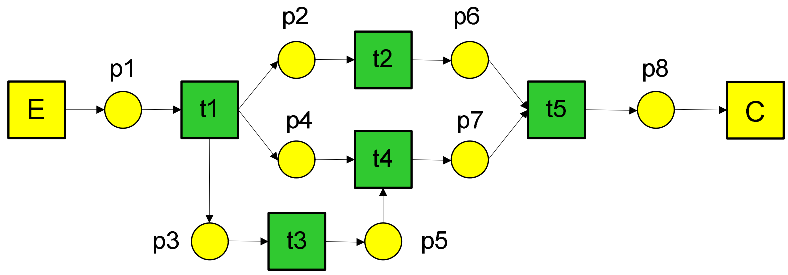


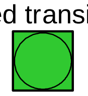
Figure 7: Case sensitive places

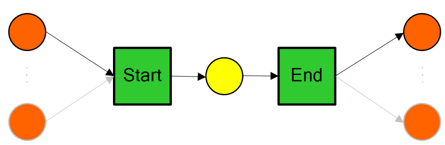
Transition *t4* in [Figure 7] is enabled for firing, when there are tokens with the same case id in *p4* and *p5*. When it fires it will produce a token with the same case id in *p7*.

Inhibitor arcs and reset arcs
Yasper support the use of inhibitor arcs and reset arcs, mentioned in the previous section.

Time
For each transition an expected (or average) processing time and a standard deviation of the processing time can be set. A timed transition has a circle in its body; see below. A timed transition is an abbreviation of a sub process which contains two transitions connected with a single arc to one place:

Timed transition:





The time is modelled in tokens. The time is modelled as a timestamp of a token. Tokens may only be consumed if the current time is greater or equal to the timestamp of the token. The first transition creates the new timestamp for the token; the token remains in the place until the second transition can consume the token.

Roles
Roles can be assigned to a transition to define the human resources or machinery the transition needs. A transition can only be executed when enough resources are available. When all the resources are occupied, the case must wait until the resources are released. The simulation function can calculate the processing, the waiting and the throughput time for a case.

Hierarchy
With the subnet element, Yasper supports hierarchy. Within a flat Petri net, a subnet can be created for each transition bounded selection. Inside the subnet reference places are used to refer to the places outside the subnet. The reference places show the connections with the process on a higher level. Reference places can not contain tokens. A transition with a reference place as input place is enabled for firing when the place referred by the reference place has a token. When a transition has a reference place as output place, it will produce a token in the place that is referred by the reference place.

In [Figure 8] a part of a Petri nets is shown. It contains a subnet called “Subnet”. The subnet has two input places (p1 and p2) and one output place (p3).

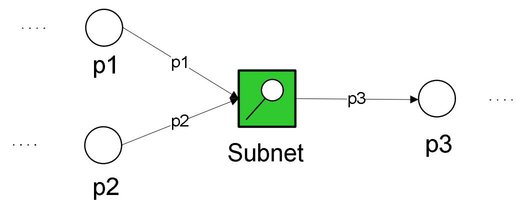


Figure 8: A subnet

When we expand the subnet, we see two input reference places, which refer to p1 and p2.

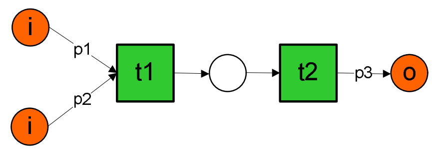


Figure 9: A collapsed subnet

In [Figure 9] transition *t1* is enabled for firing when there is a token in *p1* as well as in *p2*. When transition *t2* fires, it will produce a token in *p3*.

<u>XOR</u>
Yasper contains a transition of type *xor* or choice. It consumes a token from **one** input place and produces a token in **one** output place. The semantics of a transition of type *xor* are the same as a net of transitions with one input and one output arc, all connected by one place in the middle of them:

When a xor in a case-sensitive process consumes a case token from a case-sensitive place, it must produce a token with the same case id in a case-sensitive output place.

<u>The rules for execution of transition</u>
All tokens in case-sensitive places have a case id. Tokens of the same case have the same case id. Transition execution works on a case-by-case basis: all tokens consumed from case sensitive places must be of the same case. The produced tokens in the output places have the same case id as the consumed ones.

Execution of a transition with respect to case *c* can start when:

- All roles assigned to the transition are available
- All normal places connected to the transition with an inhibitor arc are empty and no case sensitive places connected with inhibitor arcs to the transition contain any tokens of case *c*
- Every input place contains as many tokens as there are arcs from it to the transition, all of case *c* in the case sensitive places

When the execution of a transition starts, the following happens atomically:

- A set of tokens on the input places as just described is consumed
- All roles that are assigned to the transition are occupied
- All normal tokens are removed from normal places connected with a reset arc to the transition
- If any case token is consumed, all tokens of case *c* are removed from the case sensitive input places connected to that transition with reset arcs

- The processing time of the transition is determined

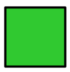
When the processing time of the job has expired, the following happens atomically:

- If the transition occupied a role, the role is released
- If starting the transition consumed any case tokens, for each case sensitive output place as many case tokens of case *c* are produced in the case sensitive output place as there are arcs from the transition to the case sensitive place; and for each normal output place as many normal tokens as arcs from the transition to the normal output place are produced
- If starting the job did not consume any case token, no token is added to any case sensitive output place.

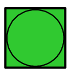
1.2.1. Summary of the building blocks

The process models in Yasper are built up with the following elements:

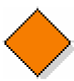
Transitions



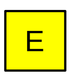
<u>Normal transition</u> Timing information is not set, the execution of this transition takes no processing time.



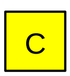
<u>Timed transition</u> The mean processing time and mean deviation are set.



A transition of type “XOR” is enabled when there is at least one token in an input place. It takes one token from an input place and produces one token in an output place. Probabilities for each output place can be set for automatic simulation.



An <u>emitor</u> creates *case* and normal tokens at a specific rate on all its outgoing arcs. Case tokens are produced for each case sensitive output place and normal tokens otherwise. The mean firing rate and deviation can be set. Because it is a starting point of a process it can only have outgoing arcs.

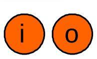


A <u>collector</u> consumes the tokens in its input places with the same rules as a transition and removes the tokens from the process. A collector is an ending point of a process; it can only have incoming arcs.

Places

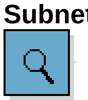
A <u>normal place</u> can only hold normal tokens.

A <u>case-sensitive</u> place can only hold case tokens.



<u>Reference places</u> are used in subnets for referring to a place at a higher level in hierarchy. Each reference place refers to exactly one place. The ‘*i*’ inside the place denotes that it is an input place for the subnet; an ‘*o*’ denotes it is an output place of the subnet. An input reference place must have one outgoing arc; an output reference place must have one incoming arc. Ambiguities can be clarified by naming: an arc to or from a subnet can contain a label, which will be shown at the reference place.

Subnets

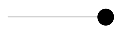


A <u>subnet</u> is a compound transition. It can contain the same elements as a normal net, except that they may contain reference places.

Arcs

A <u>normal arc</u> can contain a number also called the multiplicity of the arc. Instead of draw two arcs between the same two objects, an arc with multiplicity two can be drawn.

A bi-flow arc shows that a place is an input place as well as an output place.



An inhibitor arc must have the little ball at the transition side of the arc.

A reset arc is represented with a dotted line.

1.3. The PNML structure

Yasper uses PNML to store the processes. PNML is an XML storage format for Petri nets [PNML]. The XML structure of the PNML file is a direct consequence of the data model of PNML. In the [Figure 10], the data model of PNML is given.

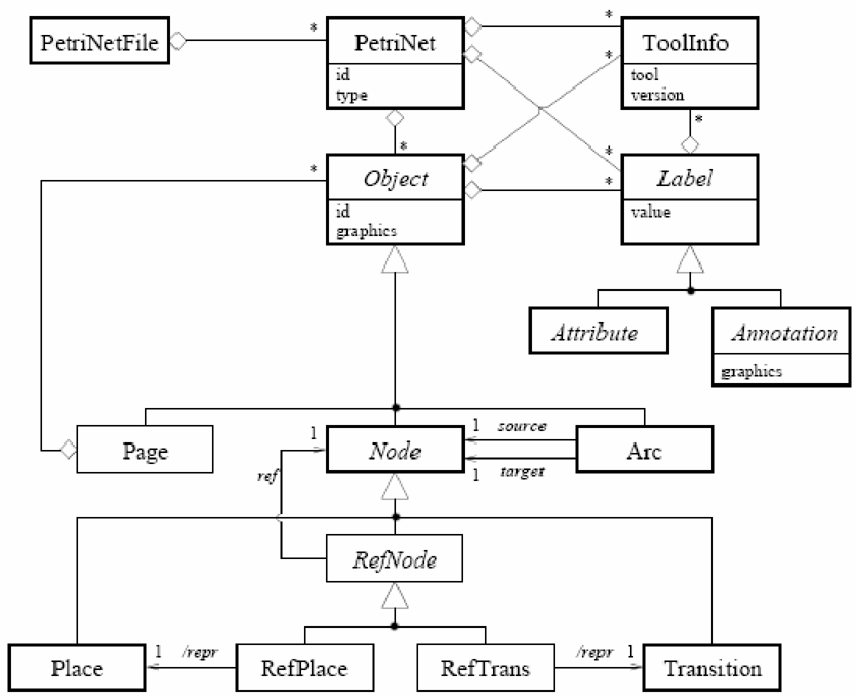


Figure 10: PNML data model

As we can see in the data model, a PetriNetFile can contain several PetriNets. A PetriNet contains several objects, like nodes (places or transitions) and subnets (pages). Nodes (transitions, places or reference objects) are linked through arcs. Every object has an *id* and contains graphical information. Every object can have a label, where extra information, such as a description or the name of the object can be stored.
The hierarchy of xml is used to store the PNML elements. A PNML-file can contain multiple Petri nets. The root of the document is the `<PNML>`-tag. The root is followed by a <net>-tag, for each disjunctive Petri net the file contains. The <net>-tag contains all the objects of the process, such as <transition>-tags, `<arc>`-tags, `<place>`-tags and <page>-tags. A `<page>`-tag, is the tag for a subnet, like the *Page* object in the data model. Such a tag can contain several other elements, including other subnets. Every element has a unique id, which will be stored as an attribute in the element tag. This *id* will be used by identifying and reference purposes. An `<arc>`-tag has, for example, a source and a target attribute. The values of those attributes are the id's of the elements which are the start respectively the end object of the arc.

Example of the xml-code for a transition**:**

A transition has an id, a name, a type and graphical information, such as its position and its dimension. Yasper may add tool-specific data such as statistical information in the `<tool-specific>`-tag. The resulting xml-code for a transition is the following:

```
<transition id="trans1">
  <name>
    <text>Transition 1</text>
  </name>
  <graphics>
    <position>
      <x>100</x>
      <y>100</y>
    </position>
    <dimension>
      <x>20</x>
      <y>20</y>
    </dimension>
  </graphics>
  <type>
     <text>and</text>
  </type>
  <toolspecific tool="Yasper" version="xxx">
    … code for statistical information
  </toolspecific>
</transition>
```

Example of a Petri net with its pnml-code:

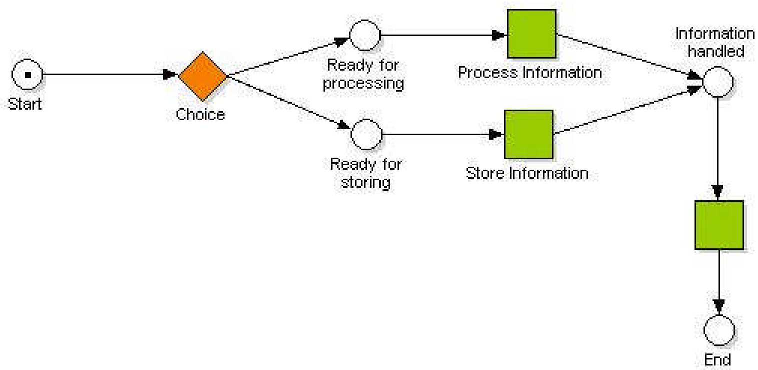


Figure 11: A simple Petri net

The xml code for the Petri net as shown above:

```
<?xml version="1.0" encoding="utf-8"?>
    <PNML>
      <net type="http://www.yasper.org/specs/ePNML-1.1" id="do1">
        <toolspecific tool="Yasper" version="xxx">
          <roles xmlns="http://www.yasper.org/specs/ePNML-1.1/toolspec" />
        </toolspecific>
        <place id="pl5">
          <name>
            <text>Start</text>
          </name>
          <graphics>
            <position x="62" y="114" />
            <dimension x="20" y="20" />
          </graphics>
        </place>
        <place id="pl6">
          <name>
            <text>Ready for processing</text>
          </name>
          <graphics>
            <position x="287" y="88" />
```

```
    <dimension x="20" y="20" />
  </graphics>
</place>
<place id="pl7">
  <name>
    <text>Ready for storing</text>
  </name>
  <graphics>
    <position x="288" y="154" />
    <dimension x="20" y="20" />
  </graphics>
</place>
<transition id="tr4">
  <name>
    <text>Process Information</text>
  </name>
  <graphics>
    <position x="398" y="87" />
    <dimension x="32" y="32" />
  </graphics>
</transition>
<transition id="tr5">
  <name>
    <text>Choice</text>
  </name>
  <graphics>
    <position x="179" y="115" />
    <dimension x="32" y="32" />
  </graphics>
  <type>
    <text>XOR</text>
  </type>
</transition>
<transition id="tr6">
  <name>
    <text>Store Information</text>
  </name>
  <graphics>
    <position x="396" y="154" />
    <dimension x="32" y="32" />
  </graphics>
</transition>
<place id="pl8">
  <name>
    <text>Information handled</text>
  </name>
  <graphics>
    <position x="522" y="119" />
    <dimension x="20" y="20" />
  </graphics>
</place>
<transition id="tr7">
  <graphics>
    <position x="523" y="214" />
    <dimension x="32" y="32" />
  </graphics>
</transition>
<place id="pl9">
  <name>
    <text>End</text>
  </name>
  <graphics>
    <position x="523" y="284" />
    <dimension x="20" y="20" />
  </graphics>
</place>
<arc id="a1" source="pl6" target="tr4" />
<arc id="a2" source="pl5" target="tr5" />
<arc id="a3" source="tr5" target="pl6" />
<arc id="a4" source="tr5" target="pl7" />
<arc id="a5" source="pl7" target="tr6" />
```

```
    <arc id="a6" source="tr4" target="pl8" />
    <arc id="a7" source="tr6" target="pl8" />
    <arc id="a8" source="pl8" target="tr7" />
    <arc id="a9" source="tr7" target="pl9" />
  </net>
</PNML>
```

2. Provision

In this chapter Provision Enterprise is introduced. What can be done with this software package? We will discuss the workflow modeler, the conceptual constructs it contains and the way the process models can be converted into PNML.

2.1. Provision Enterprise

Provision Enterprise, developed by Proforma Corporation, is one of the process modeling tools used at Deloitte. ProVision is an enterprise modeling software package, which consists of a set of strategy, process and system modelers. The main objective of this software package is to understand, analyze and improve all dimensions of the enterprise: who, what, why, where, when and how. When these enterprise concepts are understood and modeled, they collectively represent the organization, the operation and the decision-making architecture, or framework of the company.
ProVision offers a wide variety of support for process and system models. The tools available for enterprise and organization modeling are:

- Organization models
- Communication models
- Enterprise architecture models
- Location models
- Resource and cost models
- Mapping organization strategies to perform measures
- Managing process portfolios

Tools for defining business processes:
- Business class models
- Data class models
- Sequence models
- Workflow models
- Use case models
- Analysis of business processes
- Simulation of business processes

The models are stored in a shared repository, which provides standard APIs for integrating ProVision with other tools. In [Figure 12] below the architecture of Provision Enterprise is shown.

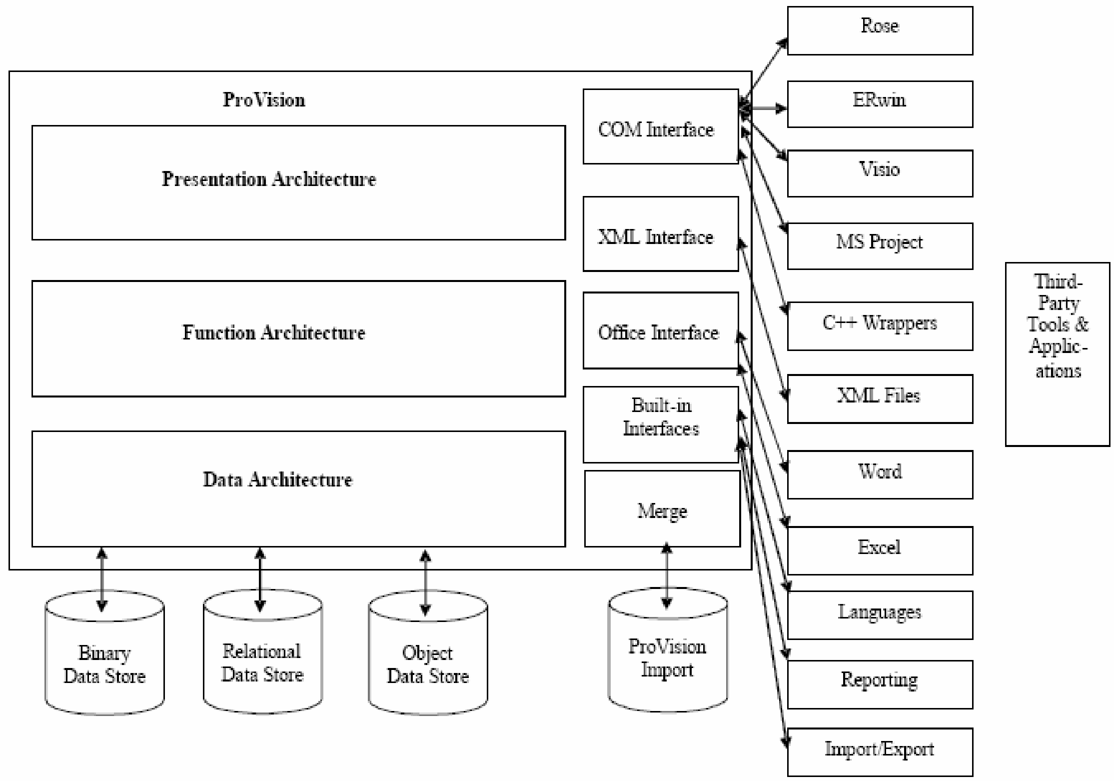


Figure 12: Overview of ProVision Enterprise architecture

ProVision relies on workflow diagrams to represent business processes, which are easier to understand for business managers than use case diagrams and class diagrams, which are more abstract. However, those diagrams can be attached to the workflow diagrams to satisfy software developers as well.

Processes can be defined top-down or bottom-up. With a top-down approach, the modeler starts modeling on the highest level of abstraction. The hierarchy is used for refining the process. The overall picture of the process is soon available, without diving into details. With a bottom-up approach, the modeler starts modeling on the lowest level of abstraction. Sub processes are used for grouping processes together. With this approach the modeler could focus on a particular part of the process in detail. Because both approaches are available, the advantages of both approaches can be used in a mixed-strategy.

2.1.1. Business processes

Within Provision processes are modeled in the workflow manager. The processes are constructed as directed graphs. The edges are the links[1], the nodes are the elements. The order of execution of the elements is defined by links. A business process can be hierarchical; a parent activity can have multiple child elements, which can be hierarchical too.

Provision Enterprise supports multiple graphical notation languages, including ProGuide (proprietary format), Rummler-Brache [Rummler-Brache] and UML [UML].

Regardless of the notation used, data from multiple diagrams that represent different perspectives of the same concept are stored in one single object. The generic support for different models is possible because the basic principles of business process modeling languages are quite similar. Those basic principles, also called concepts or meta-model elements, are basically the same, but named differently in these languages. We see tasks, which

[1] Those links are called "workflows" in Provision Enterprise 4.4 (in 5.0 they are called links). We will use the term "link".

can be refined hierarchically into sub processes or have an execution description. Resources and deliverables (including human resources, machinery) required to perform the task may also be mentioned. The order of execution, concurrency and the choice of the execution path is often given in graph like form. At last there may be facilities to describe alternatives in a process flow (decisions, branching) or to describe concurrency among tasks. Those semantics of business modeling languages are typically based on some well-known concepts of discrete mathematics, like finite state machines or Petri nets.
We recognize those concepts stated above in ProVision Enterprise:

- An activity is an elementary step or is a hierarchical element
- A link (directed arc) denotes the order of execution
- A decision point is used for branching (decision making, a choice)
- Junction: used for creating of joining parallel flows
- Resources, deliverables and roles can be set

2.2. Language constructs

A Provision process can be build with the concepts stated in the previous section. A Provision process is a directed graph. The nodes are the activities, reference elements, decision points, junctions, sources or sinks. The links are the edges. Deliverables are traveling along links. Deliverables are produced during the execution of an activity or created at a source element.

<u>Recipes</u>
The input and output relation of activities and junctions is stored in recipes. The default behavior of an activity or junction is defined by the default recipe. Custom recipes can be used to change the input and output relation. The default recipe needs all deliverables on incoming links to execute and produces the deliverable on all outgoing links: it is input and output complete.

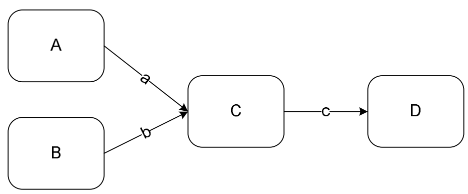


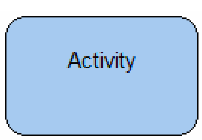
Figure 13: a Provision process

When no custom recipe is defined at activity C [Figure 13], it needs deliverable *a* and *b* to produce *c*. A custom recipe is a function which determines the inputs (deliverables) needed to product the output (deliverables). A custom recipe takes a subset of the deliverables on the incoming links as input and produces a subnet of the deliverables on the outgoing links as output. For every activity multiple custom recipes can be defined. A custom recipe looks like:

Example:
When an activity has the custom recipe: a * 2 + b * 1 → c * 1, it needs two *a*'s and one *b* to produce deliverable *c.*

The nodes we can use are:

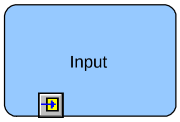
Activity



An activity represents an atomic piece of work. It takes the input deliverables it receives and transforms it into one or more output deliverables. An activity has a description-field, where textual information about the activity can be stored. Custom recipes can be set to alter the default input and output relation.
The mean processing time and the distribution of the processing time can be set, as well as the type of the queue (FIFO, LIFO, SIRO), costs based information like the direct and indirect costs of the activity and associations

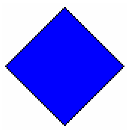
with other objects.

Reference elements



The connections of a sub process in its parent net are graphically shown by a reference element. These reference elements are the input and outputs on this level of hierarchy and must be connected with the elements in this sub process. An input can only have outgoing links and an output can only have incoming links. The name of the referred element is shown in the reference element.

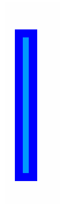
Decision point



OR join / split

A decision point evaluates a specified condition during the process and directs the flow of work based upon its evaluation. A decision point receives a single link and has two or more possible paths to subsequent nodes based on the number of alternatives for the condition. The discriminators can be visible along the outgoing links. For each possible path a probability can be set, which can be used for simulation and analyzing purposes.

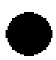
Junction



AND join / split

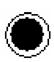
Junctions are used to combine or split deliverables. Parallel paths can be created or merged with this element. A junction must converge or diverge: the number of incoming and outgoing links must be unequal. Recipes specify the type and number of entities required to initiate the junction and the type and number of entities produced. A junction can have multiple recipes to simulate alternative scenarios.

Source



A source is the starting point of a process. A source introduces deliverables. A source emits deliverables at simulation on all outgoing links.

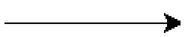
Sink



A sink is the ending point of a process.

The nodes are connected with links:

Links



Links are the passing of control from one activity to another. A link signals the completion of one activity and the initiation of the next. A link connects exactly two elements.

<u>Modeling constraints</u>

There are a couple of modeling constraints:

- A source must have one or more outgoing links and may not have any incoming links.
- An activity must have one or more incoming links and must have one or more outgoing links.
- A decision point must have one –and only one– incoming link and must have two or more outgoing links.
- A junction must have one or more incoming links and must have one or more outgoing links. The number of incoming links must be different from the number of outgoing links.
- A sink must have one or more incoming links and may not have any outgoing links.
- An input reference element must have one outgoing link.
- An output reference element must have one incoming link.

The data model of Provision is given in [Figure 14] below:

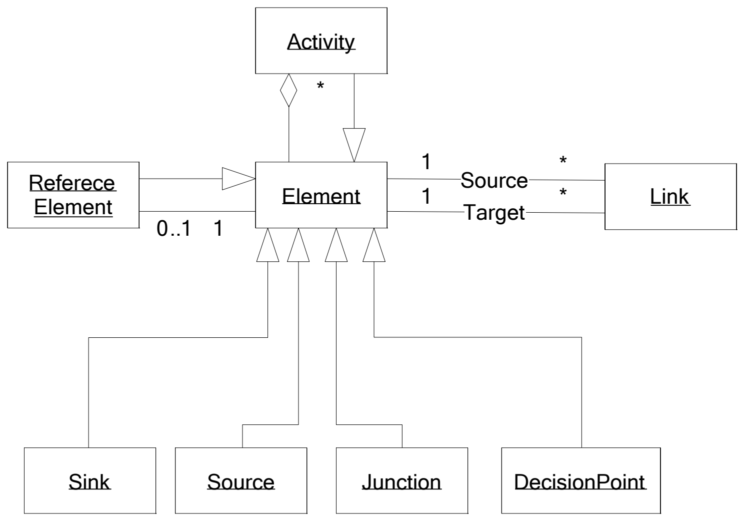


Figure 14: Provisions data model

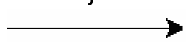
Remark: An activity is hierarchical, which is indicated by the composition diamond.

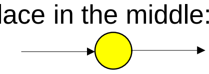
2.3. The translation into Petri nets

For the translation of a Provision process into a Petri net, the Provision process is divided into smaller pieces. Each language construct can be translated into a small net of Petri net elements. The links between the Provision language constructs and the translations of the constructs remain the same.

2.3.1. The translation of the links

A link between two objects is translated into two arcs with a place in the middle:

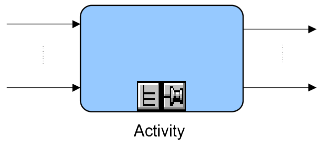




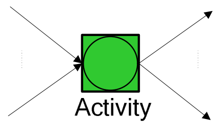
2.3.2. The translation of the nodes

A node can have several incoming and outgoing links. For the translation of the nodes, the number of incoming and outgoing links in the translation is not fixed.

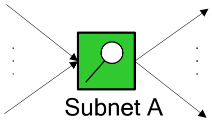
2.3.2.1. Activity



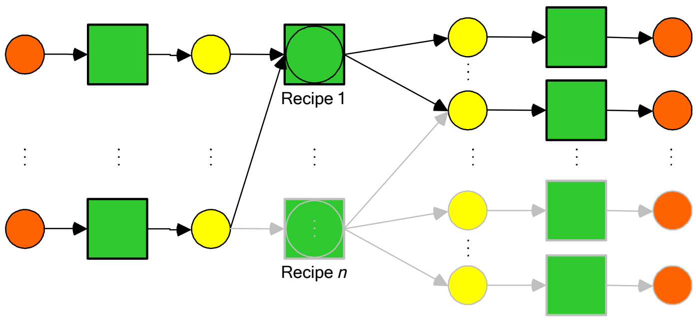
For an activity we need to take the recipes into account. Those recipes determine the behavior of the activity. Most of the time the default recipe is used: an entry for each incoming and outgoing deliverable with a count equal to the number of links containing that deliverable.
In this case the translated Yasper net will look like:



When a custom recipe is used, the activity can only be executed when all of the input deliverables of a recipe are available. When the required inputs for multiple recipes are met, only one recipe is chosen to be processed.



With Subnet A:

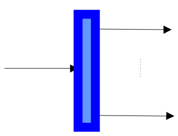


“Subnet A” has a reference place for each incoming link and a reference place for each outgoing link. The extra transition after or before a reference place is needed because a reference place can not have multiple outgoing arcs. Each link in Provision carries a deliverable. Each input and output deliverable is translated into a place. Each recipe of the activity is translated with a transition. The input places of that transition are the places belonging to the input deliverables in the recipe. The output places of that transition are the places belonging to the output deliverables in the recipe.

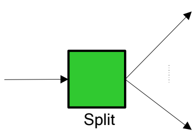
2.3.2.2. Junction

A junction has multiple outgoing links and one incoming links or visa versa. The first is called junction for branching, the latter a junction for joining.

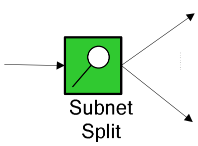
Branch:



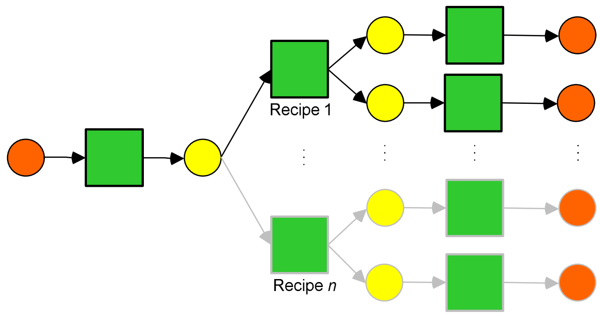
A junction can also have recipes. The default recipe has an entry for each incoming and outgoing deliverable with a count equal to the number of links containing that deliverable. In that case the junction can be translated into:



When a custom recipe is used, we need to add an extra level of detail in the translation.

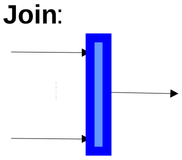


With “Subnet Split”:

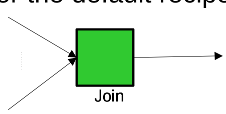


“Subnet Split” has a reference place for its incoming link and a reference place for each outgoing link. Each link in Provision carries a deliverable. Each input and output deliverable is translated into a place. Each recipe of the activity is translated with a transition. The input places of that transition are the places belonging to the input deliverables in the recipe. The output places of that transition are the places belonging to the output deliverables in the recipe.

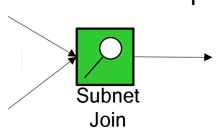
Join:



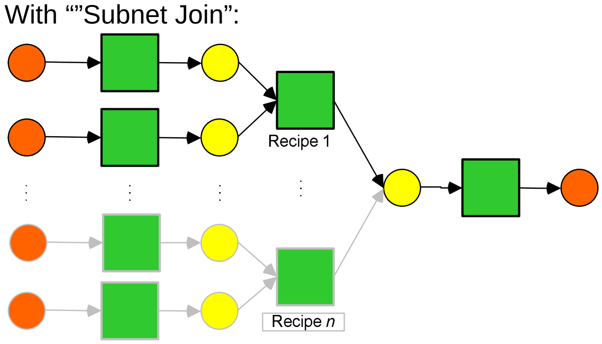
For the default recipe:



When custom recipes are used:

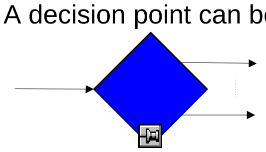


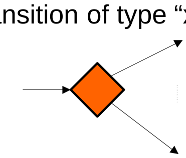
With “”Subnet Join”:



2.3.2.3. Decision point

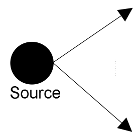
A decision point can be easily translated to a transition of type “xor”:

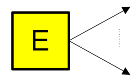




2.3.2.4. Source

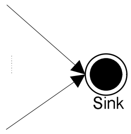
A source is the starting point of the process and can be translated into an emitor.

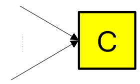




2.3.2.5. Sink

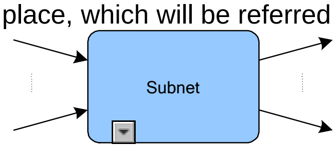
A sink is the ending point of the process and can be translated into a collector.

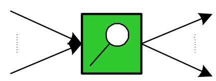




2.3.2.6. Sub processes

A sub process will be translated into a sub net. Each incoming and outgoing link will have its own place, which will be referred by a reference place in the sub net itself.





2.3.2.7. Reference Elements

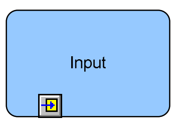


Figure 15: Input

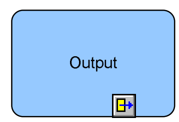
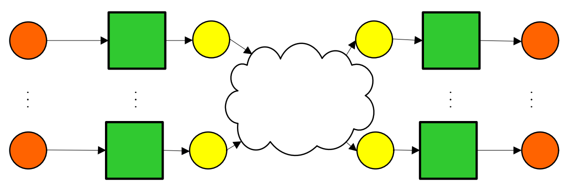


Figure 16: Output

The inputs and outputs of a sub process are inherited from its parent. In the workflow modeler the inputs are shown with the input marker [Figure 15] and the outputs with the output marker [Figure 16]. The reference elements can be translated into reference places in PNML.



2.4. Implementation

The process models created in Provision are stored in Provisions own pvw-format, which can only be opened with Provision. Provision contains an “export to xml” function, but not all data about the process are stored in this format and the xml is not readable. The use of the Provision API (Application Program Interface) gains access to all data of a process. The API is implemented in the programming language C++ and therefore my implementation is written in C++ too. By using the API we have total access to the data of every object in Provision. A disadvantage of using this method is that you must have access to Provision to translate a process with this tool. The application will open a Provision file and translate the process within the Provision file into PNML.

The translation tool starts with the creation of an object that instantiates the API class. The class must be called with some parameters, e.g. the filename of the Provision process that contains the process model that must be translated. After the initialization, we can access the methods of the API class and gather the information about nodes and links of the process through the object.

The API class contains methods to collect all nodes of a process, collect the incoming and outgoing links of an element and gather information such as the position, the name and the id of an element.

Generally, the conversion from a Provision process to a PNML file will follow these steps:

- Scan a Provision process for all of its elements with the following rules:
    - o Perform a deep-first scan starting with the net on the highest level in the hierarchy
    - o For each element:
        - Call its function to create the appropriate PNML code
        - Create the appropriate links with other elements

All the subnets on the top-level of hierarchy are collected in variable `aWorkflowModels` of type `CpvwWorkflowModels`. For each model the objects are collected with the `GetModelObjects()` function. That function returns a list of objects on the current level. The objects are checked for their type and for each type the right function is called to generate the correct PNML code:

```
CreatePNMLTransitions(aModel);
CreatePNMLSinks(aModel);
CreatePNMLSources(aModel);
CreatePNMLStores(aModel);
CreatePNMLXOR(aModel);
CreatePNMLArcs(aModel);
CreatePNMLSubnets(aModel);
```

Those functions create the corresponding PNML-code for the object. The `CreatePNMLSubnets()` function is a recursive function that creates the code for the subnet itself and calls the functions stated above for all the objects in that subnet.

3. BPMN: introduction

The Business Process Managements Initiative (BPMI) has developed the Business Process Modeling Notation. BPMN provides a graphical notation and semantics of a Business Process Diagram (BPD) in an informal way. The primary goal of the development of a new process modeling notation was to bridge the gap between the process design and process implementation. Another goal was to visualize xml languages designed for the execution of processes, like BPEL4WS [BPEL4WS].
The BPMI is a working group which is composed of several prominent companies like IBM, PeopleSoft, Lombardi Software, Popkin Software and Proforma Corporation. They have expertise and experience with several existing modeling notations and they sought to consolidate the best ideas from different notations into one single standard notation. In the academic world Petri nets have been used for many years to model complex processes graphically. Petri Nets have a formal basis and many terms of BPMN are drawn from Petri nets. Other methodologies and notations that were reviewed are e.g. UML Activity Diagrams [UML AD], UML EDOC Business Processes [EDOC], IDEF [IDEF] and Event-Process Chains [EPC].

In recent years, there has been a lot activity in developing web services-based XML execution languages for Business Process Management systems. Languages such as BPEL4WS provide a mechanism for the definition of processes. Those languages are tuned for the operation and inter-operation of BPM systems. The format is very well handled by a software system, but hard to work with for designers, analysts and managers. The BPMI tries to bridge this technical gap by mapping a standard visualization notation (BPMN) to an appropriate execution format (BPEL4WS).

3.1. BPMN language constructs

In the official BPMN specification there are four groups of modeling elements: flow objects, connecting objects, swimlanes and artifacts.

The first two groups, the flow objects and the connecting objects, are used for defining the process. Swimlanes and artifacts are used to give extra information about the process model to the reader.

<u>Flow objects</u> are the main elements to define the behavior of a process. There are three flow objects: *activities (tasks and sub processes)*, *gateways* and *events*.

<u>Connecting objects</u> are arcs which connect elements with each other: *sequence flows* are used to connect flow objects to show the order of execution, *message flows* show the communication between processes and *association flows* connect flow objects, connecting objects or swimlanes with artifacts. Association flows do not play a role in the semantics: these are only for documentation purposes.

<u>Swimlanes</u> are used to visually separate activities in order to illustrate different functional capabilities or responsibilities. There are two kinds of swimlanes: pools and lanes. A pool represents an entity like an organization. Pools can be organized or categorized by lanes; a pool can be horizontal partitioned by lanes. A lane can be for example a department in that company. Lanes do not have a semantic meaning.

<u>Artifacts</u> are used for documentation. Artifacts show additional information about the process. There are three kinds of artifacts: *Data objects* represent the documents in the process that activities require and/or produce. *Annotations* are comments about the process that can be linked by association flows to flow objects or connecting objects. *Groups* can be used to group activities (over pool and lane boundaries) in order to add comments about the members of the group.

From the official specification [BPMN] we derive the following more precise definitions:

A process is the largest coherent component of a graph where the nodes are *activities*, *gateways* or *events* connected to each other with arcs that are *sequence flows*. A process has one start event and one end event.

A sub process is a process which is a part of another process; it behaves like a task, but it contains a process. When we replace the sub process with its contents, with the rules described in [Appendix C], the resulting process is a normal process again. Two nodes are on the same process level if they occur in the same sub process.

A system is a disjoint of collection processes, connected with arcs that are *message flows*, used to exchange messages between the processes.

Each process belongs to exactly one pool; *message flows* are always between *tasks* or *events* of different pools.

3.2. Example

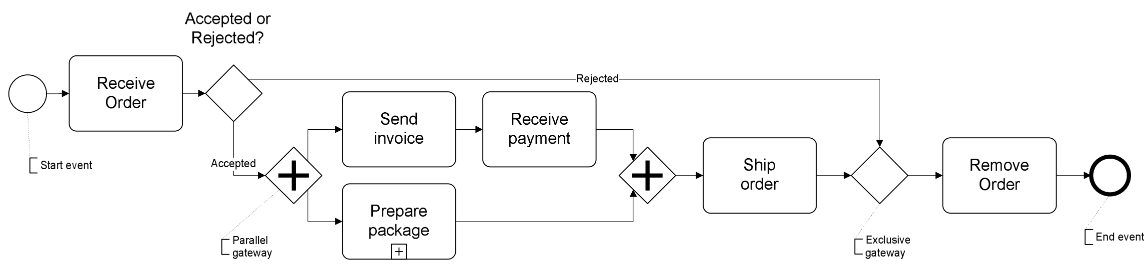


Figure 17: A Business Process Diagram (BPD)

In [Figure 17] the receipt and transaction of an order is modeled. The process starts with a starting point: a start event. The process continues when an order is received. Depending on the status of the order, rejected or accepted, a different path will be followed. When the order is accepted, the process will continue on two paths in parallel (both paths of the parallel gateway will be followed). On the first path, an invoice will be sent followed by the receipt of the payment and the second path is the preparation of the package. The paths will be joined with a parallel gateway when those activities are completed and the order will be shipped. It will pass the exclusive gateway and eventually the order will be removed, resulting in reaching the end event: the completion of the process.
When the order is rejected, the order will pass the exclusive gateway, the order will be removed and the process will be completed as well.

3.3. Overview of the BPMN objects

In this section the syntax of the objects is given. The objects that define the process will be discussed in more detail in [Section 4.2].

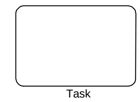
3.3.1. The nodes

3.3.1.1. Activities

There are two types of activities: tasks and sub processes. Both types share the same sub types: *loop*, *multi-instance loop* and *compensation*. The sub types can be distinguished by markers centered on the bottom of the activity symbol. See [Section 4.2.1] for more details.

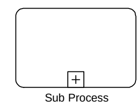
Task

A task represents an atomic piece of work in the process. A task is represented by a rounded-corner rectangle.



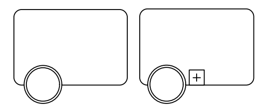
Sub Process

A sub process contains a process and can be collapsed to see its content. A sub process can be distinguished from a task by the sub process marker $\boxplus$.



Interruptible activities

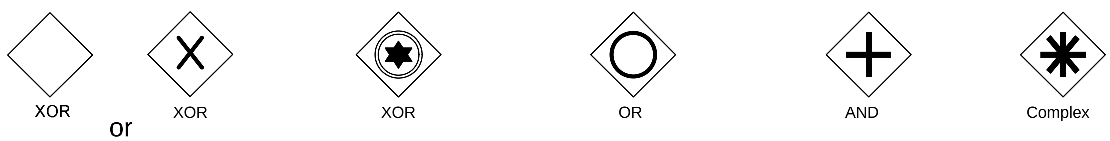
In this paper, activities with an intermediate event on their boundary are called interruptible activities. The intermediate event is called the interrupt event and the sequence flow outside the interrupt event is called the exception flow of the interruptible activity.



3.3.1.2. Gateways

A gateway is a point in the process where alternative or parallel paths are created or joined. A gateway is represented by a diamond shape. There are five types of gateways, each with their own behavior. They can be distinguished by a different internal marker.

Data-based exclusive (XOR)	Event-based exclusive (XOR)	Inclusive (OR)	Parallel (AND)	Complex



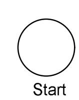
For the data-based exclusive gateway the modeler must choose one of the shapes. According to the official specification, gateways may have multiple incoming sequence flows and multiple outgoing sequence flows at the same time. However, we assume that all gateways have one incoming sequence flow and multiple outgoing sequence flow, or visa versa. Depending on the type of the gateway one or more paths will be followed. See [Section 4.2] for more information.

3.3.1.3. Events

Events are the starting and ending points of a process or indicate that something happens during the execution of a process. An event is represented by a circle. Events are circles with open centers to allow internal markers to differentiate different triggers or results. There are three types of events, based on when they affect the flow:

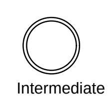
Start events

A start event is represented by a single-lined circle. They will indicate where the (sub) process starts.



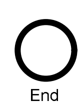
Intermediate events

Intermediate events are represented by a double-lined circle. They occur between the start and the end event of a process.



End events

End events are represented by a bold-lined circle. They indicate where a (sub) process ends.



There are nine internal markers: Message, Timer, Error, Cancel, Compensation, Rule, Link, Multiple and Terminate, each which its own symbol. In [Section 4.3] these types are discussed in more detail.

3.3.2. The edges

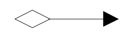
The nodes of BPMN are connected with each other through arcs.

3.3.2.1. Sequence flow

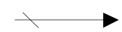
Sequence flows are used to connect two nodes and show the order of processing. A sequence flow can not have multiple flows between the same objects. There are four types of sequence flows: *normal flow*, *conditional flow*, *default flow* and *exception flow*. The different types are distinguished by a different symbol on the tail of the arc or by the position where the flow occur.

Normal flow
A normal flow has no conditions; when the start node is completed the flow continues along this arc.

Conditional flow
Every conditional flow has a conditional expression that is evaluated at run-time to determine if the flow continues along this arc. A conditional flow is often used in combination with a default flow. When the start node is a gateway, the mini-diamond is not shown.



Default flow
A default flow must be used together with the conditional flow. The default flow will be followed when all the conditional flows of the start node evaluate to false.



Exception flow
The outgoing flow of an interrupt event of an interruptible activity is called an exception flow.

3.3.2.2. Message flow

Processes can send messages to each other along message flows. Messages can be used to synchronize or start processes. A message can only travel on message flows. Not every node can send or receive messages. See [Section 4.4] for the nodes which are allowed to send and/or receive messages.

A message flow is a directed arc. A message flow can not connect two objects within the same pool and because each pool contains a disjoint process, it only connects objects from disjoint processes.

3.3.2.3. Association flow

An association flow can be used to connect artifacts with nodes or edges.

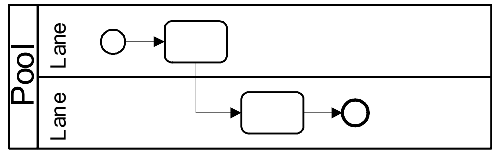
A directional association is often used with data objects to show that a data object is either an input to or an output for an activity or gateway.

3.3.3. Swimlanes

BPMN supports swimlanes with two main constructs.

3.3.3.1. Pools and Lanes

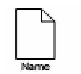
A pool is a container for a process. A pool can be partitioned into lanes. Sequence flows can not cross the pool boundary but can cross the lane boundary. The figure alongside shows an example of a simple process with swimlanes.



3.3.4. Artifacts

3.3.4.1. Data Object

Data objects provide information about what documents an activity requires and/or produces. A data object is connected to a node with an (directed) association flow.



3.3.4.2. Group

This element can be used to group flow objects for documentation purposes.

3.3.4.3. Annotation

An annotation is a mechanism to provide additional information to the reader (a comment). An association flow is used to link the annotation to the connecting or flow object.

3.4. Informal semantics

Here we define the semantics of process models. The following structural notions will not be represented in the semantics: association flows, annotations, lanes and artifacts, because these objects do not define the process.

A state of a system is a configuration of tokens. A token is an object that may reside inside a task, an event or a gateway, or a token may reside on an arc. There are two kinds of tokens: message tokens which only travel on message flows among processes and case tokens which only travel on sequence flows within the same process. The set of all possible states of the process is called the state space.

A system determines a transition system; i.e. a state space and a transition relation.

A case is a product (or service) being created in a process, from the first concept till the final result. A set of case tokens marks the stage of one case. These tokens carry the identity of the case: the case id.

Message tokens can only be sent or received by special tasks or events, see [Appendix A].

An event is triggered when it receives a case or message token. The type of the token depends on the type of the event. See [Section 4.3].

Interruptible activities have one or more interrupt events. Each interrupt event has **one** outgoing sequence flow, called the exception flow. The interrupt event can be triggered when the activity where it belongs to has an internal case token inside the activity. Each interrupt event is triggered by an intermediate event at the same process level.

An end event can have a result. The result will be send along a result flow when the (sub) process is completed. The result flow can be a message or a sequence flow and the result can respectively be a message or case token with the same case id as the consumed case token(s), all depending on the type of the end event.

A transition from one state to another state is performed by a task, event or gateway in de following way:

- A task consumes a case token from **one** incoming sequence flow and puts it inside the task.
- If the task is of type [Service, User] it sends **one** message token on the outgoing message flow and waits for a message token to be received.
- If the task is of type [Send] it sends a token on the outgoing message flow, consumes the internal case token and produces a token on **all** outgoing sequence flows.
- If the task is of type [Receive, Service, User] it is waiting for a message token from another process. If the message token on the incoming message flow arrives, it will consume the internal case token and produce a case token for each outgoing sequence flow.
- If the interrupt event of an interruptible activity is triggered, it consumes the internal case token of the activity and produces a case token with the same case id on its exception flow. If the activity is a sub process, all case tokens of the same case are removed from the process.
- A task consumes the internal case token and produces case tokens with the same case id as consumed on **all** of its outgoing sequence flows.
- A sub process consumes a token on **one** incoming sequence flow, puts it inside the sub process (this is the internal token) and puts a case token with the same case id as consumed in the start event of the process.

- A gateway consumes **one or more** case tokens from its incoming sequence flows (depending on the type and gateway expression) and puts a case token inside the gateway. All case tokens consumed by a gateway have the same case id.
- A gateway consumes the internal case token and produces case tokens on **one or more** outgoing sequence flows (depending on the type and gateway expression). All produced case tokens have the same case id as the consumed one(s).
- An intermediate event consumes a case token from its only incoming sequence flow and puts it inside the event.
- If an intermediate event of type Message or Timer is triggered, it consumes the internal case token and produces a case token with the same case id as consumed on its outgoing sequence flows.
- An intermediate event consumes the internal case token, triggers its counterpart interrupt event and produces a case token with the same case id on its outgoing sequence flows.
- If a start event is triggered, it consumes the internal case token and produces a case token with the same case id as consumed on each outgoing sequence flow.
- If an end event has consumed **all** case tokens of a case it will consume the internal case token of the sub process, it will produce its result on its result flow and it will produce a case token with the same case id as consumed on all outgoing sequence flows of the sub process, atomically.

A <u>process log</u> contains process information: i.e. the case history of each case and all the firing sequences of the nodes. The tasks, gateways and events can inspect the processlog at runtime and can use process log information for decisions.

Each task has a <u>processing time</u>, which is the time a case token is inside the task.

4. BPMN: Language constructs in detail

In this section the nodes are discussed in more detail.

4.1. Activities

4.1.1. Tasks

A task is an atomic action. A task has some attributes which can be set. It can have one or more[2] of the following markers:

Type	Description	Marker
Loop	A standard loop has a Boolean expression that is evaluated during every cycle of the loop. If the expression is still true, the task will be executed once again. The programming constructs <u>while</u> and <u>until</u> are supported by setting the *testTime* attribute to *before* or *after*.	↺
Multi-instance	A multi-instance loop reflects the programming construct <u>foreach</u>. A numeric expression is evaluated only once before the loop starts and will specify the number of times the task will be executed. The executions can be performed *sequentially* or in *parallel*, depending on the *MI_Ordering* attribute.	**II**
Compensation	Compensation can be used to roll back activities that already have completed, when an error arises in the process. The activity that has this internal marker will be executed when the interrupt event of the interruptible activity where it is connected to is triggered.	◀◀

<u>Sequence flows</u>
A task can have multiple incoming and outgoing sequence flows. Every token that arrives starts the task. When the task is completed a token will be produced on **all** its outgoing sequence flows. If the task is interruptible and the interrupt event is triggered, only **one** token will be produced on the exception flow of the task and **no** tokens on the other sequence flows.

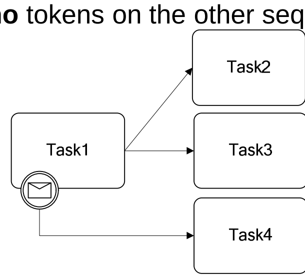


Figure 18: Tasks

<u>Example</u>:
See [Figure 18]. When Task1 is completed without interrupt, Task2 as well as Task3 will receive a case token on their incoming sequence flows. When the interrupt event is triggered a case token will be sent along the exception flow.
There are eight types of tasks, the *taskType* attribute stores the type of the task. There is no graphical distinction between them. The presence of message flows is determined by the type. See [Section 4.4] for detailed information about message flows.

4.1.2. Sub processes

We assume that every sub process has exactly **one** start event and **one** end event. Every process with more than one start or end event can be remodeled to meet these requirements. For

[2] A Loop marker may be used in any combination with the other markers except the multi-instance marker. A compensation marker may be used in any combination with any of the other markers.

more information about remodeling (sub) processes to meet these requirements see [Appendix D].

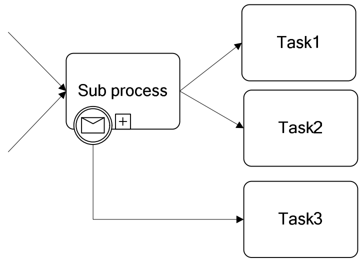


Figure 19: A sub process

A sub process starts with respect to case *c* when:

- A case token of case *c* arrives at **one** incoming sequence flow of the sub process, triggering the start event of the sub process

A sub process produces ends with respect to case *c* when all tokens of case *c* in the sub process are consumed by the end event or when the sub process is interruptible and the interrupt event is triggered by a case token of case *c*.

When a sub process ends with respect to case *c*:

- If the end event has consumed all case tokens of case *c*, it produces simultaneously:
    - A case token of case *c* on each outgoing sequence flow of the sub process
    - Its result, if it has one: a case token or message token depending on the type of the end event on the result flow of the end event
- If the sub process is interruptible and the interrupt event is triggered it **only** produces:
    - A case token of case *c* on the exception flow of the interruptible sub process.

The sub process in [Figure 14] starts when it receives a case token on an incoming sequence flow. If a message arrives during execution of the sub process, the sub process will be interrupted and will only produce a case token on the exception flow, triggering Task3. When the sub process ends without being interrupted, it will produce a case token on both outgoing sequence flows of the sub process.

A sub process can have the same internal markers as a task. See the previous section for more information about the *loop*, the *multi-instance* and the *compensation* marker. A sub process may also have the *ad-hoc* marker[3]:

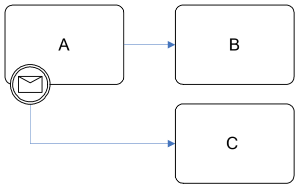
Type	Description	Marker
Ad hoc	The activities in the sub process with the ad-hoc marker do not have incoming or outgoing sequence flows. The activities can be executed in any order and as many times as needed. The attribute *AdHocCompletionCondition* is an expression that determines when the process ends.	~

4.1.3. Interruptible activities

Interruptible activities can be interrupted when its interrupt event is triggered. An interruptible activity has an extra outgoing sequence flow: the exception flow. The interruptible activity can only be triggered during execution of the activity. When the event occurs, the internal token of the activity is consumed and a case token with the same case id as the consumed internal case token will be produced on the exception flow.

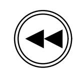
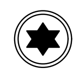
Example of an exception:

[3] The ad hoc marker may be used in any combination with the other markers



During execution of task A, the interrupt event may occur. When a message is received during the execution of task A, it will be interrupted and a case token will be send to task C. When task A is not interrupted, task B will be executed.

There are seven types of interrupt events each with its own marker:

Type	Description	Marker
Message	When the message token arrives, the internal case token of the activity will be consumed and a case token with the same case as the internal token of the activity will be produced on the exception flow.	
Timer	This event is triggered at a specific time-date or at a specific cycle.	
Error	This type will catch a (named) error (case token) send by another intermediate error event.	
Cancel (only at transactions)	This event may only be used at transactions; we do not support transactions.	
Compensation	This event is triggered when it receives a case token from an intermediate compensation event.	
Rule	This type of event will be triggered when a rule becomes true. A rule is an expression that evaluates some process data.	
Multiple	This means that there are multiple ways to trigger this event. Only one of them is required.	

4.2. Gateways

Gateways are modeling constructs that are used to control how sequence flows interact as they diverge and converge. According to the official specification, a gateway can be used for joining and splitting the flow at the same time; it can have multiple incoming and outgoing sequence flows simultaneously. However, we recommend the use of one of those functions at the same time (if possible): a gateway has one incoming sequence flow and multiple outgoing sequence flows or multiple incoming sequence flows and one outgoing sequence flow. For the evaluation of the expressions in the gateway or on the sequence flows the processlog can be used.

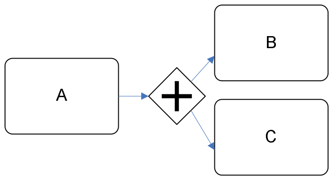
4.2.1. Parallel Gateways (AND)

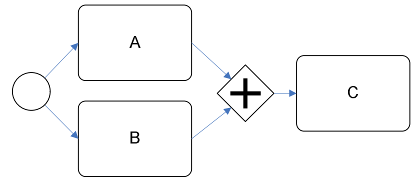
Parallel gateways are used to create and join parallel flows. They are not the only way to create parallel flows (the outgoing sequence flows of a task are always parallel), but using parallel gateways to create parallel flows will provide a balanced approach where splitting and joining can be paired up.

A parallel gateway used for splitting the flow will consume a case token on the incoming sequence flow and produce a case token with the same id on all of its outgoing sequence flows. A parallel gateway used for joining parallel flow consumes one token with the same case id from all its incoming sequence flows and produces one token with the same case as the consumed ones on its outgoing sequence flow.

Example: Create parallel flows

Example: Join parallel flows





4.2.2. Exclusive Gateways (XOR)

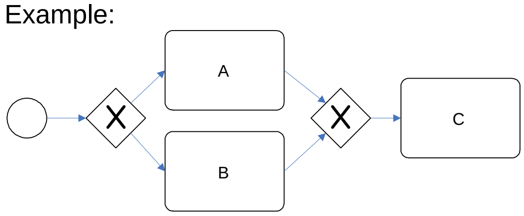
4.2.2.1. Data-based exclusive gateways

Excusive gateways with one incoming sequence flow and multiple outgoing sequence flows are used for a decision in the process. Each outgoing sequence flow has an expression that must be evaluated. The expressions are evaluated in a specific order. The first condition that evaluates to true will determine the path that will be taken. A default gate may be used to ensure that at least one of the sequence flows is chosen.

Exclusive gateways with multiple incoming sequence flows and one outgoing sequence flow are used for joining alternative sequence flows. The gateway takes a token from **one** of its incoming sequence flows and produces a case token with the same case on the outgoing sequence flow.

Remark: It is rarely required for the modeler to use an exclusive gateway for joining alternative paths. In the example below the second exclusive gateway can be omitted, because after the first exclusive gateway only one of the tasks (A or B) will be executed.

Example:



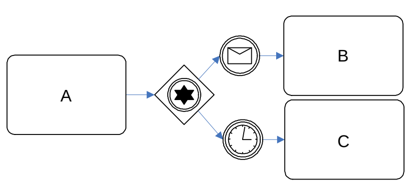
The first exclusive gateway is used as a decision, the second for joining the alternative paths.

Figure 20: A data-based exclusive gateway

4.2.2.2. Event based exclusive gateways

An event-based exclusive gateway is used in special cases in the process, where the process is waiting for the triggering of one event out of multiple events. The nodes to which the gateway is connected must be intermediate events or tasks waiting for a message token to arrive. A case token is produced on the sequence flow of the first event that is triggered.

Example:



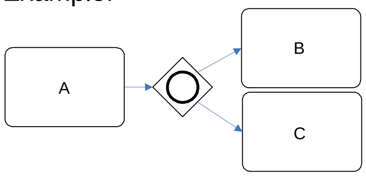
After the completion of task A the process is stalled until a message arrives or the timer is triggered. Only the first event that occurs has an impact on the process.

Figure 21: An event-based exclusive gateway

4.2.3. Inclusive gateways (OR)

An inclusive gateway with one incoming sequence flow and multiple outgoing sequence flows is used for a decision where **one or more** alternative paths can be taken. Each outgoing sequence flow has an expression. A case token with the same case as the consumed one is produced for each outgoing sequence flow where the expression evaluates to true. The modeler can use a default sequence flow that will be chosen when all expressions evaluate to false.

Example:

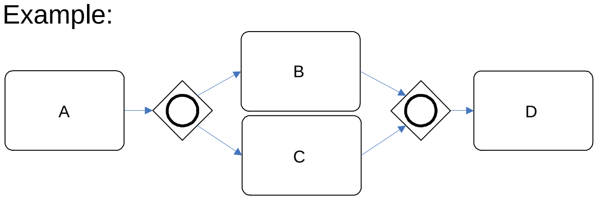


After the completion of task A the expressions on the outgoing sequence flow of the inclusive gateway will determine if a case token is produced on the outgoing sequence flow. At least one of the paths must be chosen, but also both paths could be taken.

Figure 22: An inclusive gateway

An inclusive gateway with multiple incoming sequence flows and one outgoing sequence flow is used for joining alternative flows. It will wait for all case tokens of the same case that are produced upstream. It does not require that all incoming sequence flows contain a case token. When for example an inclusive gateway downstream has produced two out of three tokens, the inclusive gateway will continue with two case tokens, contrary to the three incoming sequence flows.

Example:



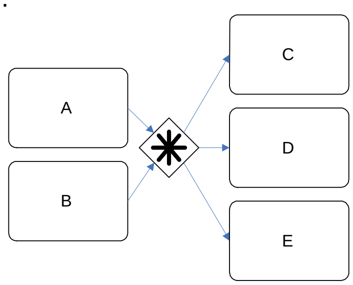
The second inclusive gateway will proceed when all case tokens produced by the first have arrived.

Figure 23: Another inclusive gateway

4.2.4. Complex gateways

A complex gateway is a construction where complex splitting and joining of paths can be handled. A complex gateway can have multiple incoming and outgoing sequence flows at the same time. It contains a (complex) expression that determines the merging and/or splitting behavior of the gateway. The expression should be designed in such a way that the process is not stalled. The expression could contain process data.

Example:



The evaluation of the expression of the complex gateway could determine to proceed with task C and task D when a case token is received from task B.

Figure 24: A complex gateway

4.3. Events

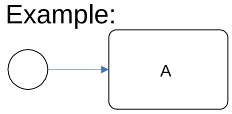
Events affect the sequence or timing of activities in a process. The three main types of events - *start events*, *intermediate events* and *end events* - occur on different places in the flow. Start events are the starting points of a (sub) process and end events are the ending points of a (sub) process. Intermediate events may occur between the start and the end event of a (sub) process.

Start events and intermediate events can have *triggers* that define the cause of the event e.g. the arrival of a message, a rule becomes true or a certain amount of time has elapsed. An end event may define a result that is a consequence of a sequence flow ending. The type of the event, shown by an internal marker, defines the trigger or result of a gateway.

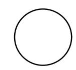
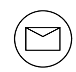
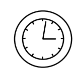
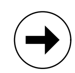
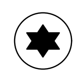
4.3.1. Start Events

A start event is the starting point of a (sub) process. A start event may not have any incoming sequence flow and must have one or more outgoing sequence flows. When a start event is triggered, it produces tokens on all its outgoing sequence flows. There are six types of start events. The type of the event will define the trigger that starts the (sub) process.

Example:



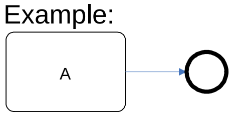
The start event starts task A when it is triggered.

Trigger	Description	Marker
None	The modeler does not display the type of the event. This type is also used for triggering a sub process when a case token arrives at an incoming sequence flow of the sub process.	
Message	A message token arrives from a participant on a message flow.	
Timer	A specific time-date or a specific cycle can be set that will trigger this start event.	
Rule	This type of event is triggered when a specific rule becomes true.	
Link	The result of an end event of another process is received and triggers this start event.	
Multiple	There are multiple ways to trigger this event, only one of them is required to trigger the event.	

4.3.2. End Events

An end event is the ending point of a (sub) process. It has no outgoing sequence flows; however it can trigger intermediate events depending on the result of the end event. In the informal semantics, the result - a case of message token, depending on the type of the end event – will be sent along the *result flow*. The result flow is not visible in the BPMN process model.
The end event consumes all the case tokens it receives on each incoming sequence flow. When **all** case tokens of a specific case are consumed, it produces a result, if it has one and it produces case tokens with the same case id as the consumed one(s) for each outgoing sequence flow of the sub process where it belongs to, atomically. The process is completed for a case if the end event on the highest level of hierarchy has consumed all the case tokens for a case.

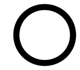
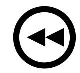
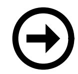
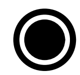
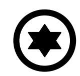
Example:



The case token that is produced at the completion of task A will be consumed by the end event. This end event generates no result.

In the table below the results with the internal marker an end event can produce are given:

Trigger	Description	Marker

None	The modeler does not display the type of the event. This type is also used to show the end of a sub process.	
Message	A message token will be send to a participant.	
Error	A case token will be send to an intermediate error event.	
Cancel	This type of event is used in a transaction. We don't support transactions.	
Compensation	This type of end will indicate that compensation is required. It will send a case token to an intermediate compensation event.	
Link	A link is a mechanism to connect the end of a process to a start of another; it sends a case token to a start event of another process.	
Terminate	This type of event indicates that all activities in the process should be ended without exception or compensation handling.	
Multiple	There are multiple consequences of ending this event; a case token (or a message token for each message) is produced for each of them.	

4.3.3. Intermediate Events

This type of event occurs after a process has been started. It will affect the flow of a process, but it will not start or directly end a process. Intermediate events can be used to:

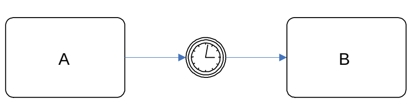
- Show where messages are expected
- Show where delays are expected in the process
- Disrupt the normal flow through exception handling
- Show extra work for compensation (compensation is extra work that must be done after the occurrence of an exception)

We distinguish two kinds of intermediate events. The term <u>intermediate event</u> is used for an intermediate event that occurs between two other nodes and the term <u>interrupt event</u> is used for an intermediate event that is used for exception handling for an interruptible activity.

4.3.3.1. Intermediate events

An intermediate event must have one incoming sequence flow and one outgoing sequence flow. An exception to this rule: intermediate link events must have one incoming sequence flow and no outgoing or visa versa.

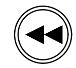
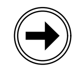
Example of an intermediate event:



When task A is completed, the intermediate timer event will wait until the event is triggered. When it is triggered it will produce a case token with the same id as consumed along its outgoing sequence flow.

There are six types of intermediate events:

Type	Description	Marker
None	The modeler does not display the type of the event. It indicates some change of a state in the process.	

Message	The process will continue when the message arrives.	
Timer	A specific time-date or a specific cycle can be set that will trigger the event. It is a delay mechanism.	
Error	This type of event will throw a (named) error, which will be caught by an intermediate event at the boundary of the sub process.	
Compensation	This event calls for compensation. It will trigger an intermediate compensation at the boundary of the sub process.	
Link	Paired link events can be used as “Go to” objects. It sends a case token to another intermediate link event within the same level of the process.	

4.4. Message flows

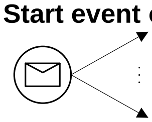
BPMN supports communication between processes by sending and receiving messages (message tokens) between disjunctive processes. The nodes must be must of the correct type to send and receive message tokens.

In the next section the nodes that can handle message tokens are given.

4.4.1.1. The communication nodes

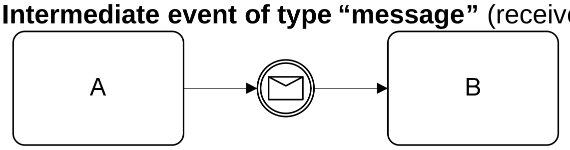
We assume that the case id of the incoming message tokens is the same as the internal case tokens which receives or sends the message token.

Start event of type “message” (receive)



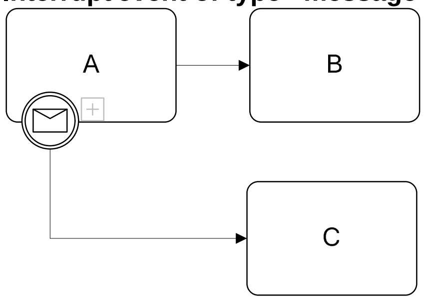
When a start event of type “message” has an internal case token, it is waiting for a message to arrive. When a message token with the same case id as the internal token arrives at the incoming message flow, it will consume the internal case token and produce one case token with the same case as the consumed one on each outgoing sequence flow. The message flow may be not shown; the internal marker already shows the type of the start event.

Intermediate event of type “message” (receive)



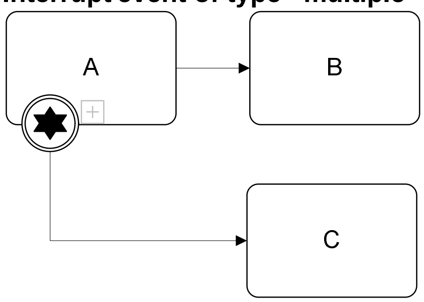
When an intermediate event of type “message” has an internal token, it is waiting for a message to arrive. When a message token with the same case id as the internal case token arrives at the incoming message flow, it will consume the internal case token and produce a case token with the same case as the consumed one on its outgoing sequence flow.

Interrupt event of type “message” (receive)



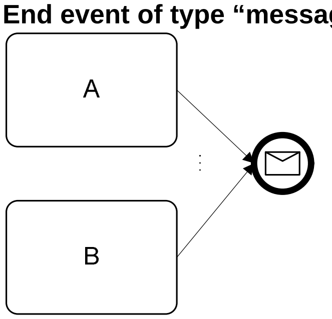
When an interrupt event of type “message” of an interruptible activity receives a message token with the same case id as the internal case token, it consumes the internal case token and produces a case token with the same case id as the consumed one on the exception flow.

Interrupt event of type “multiple” (receive)



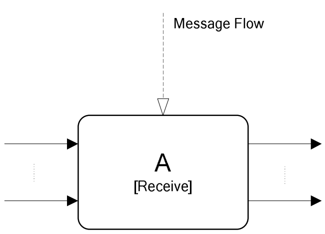
When the interrupt event of an interruptible activity is of type “multiple” and contains a message event, the interrupt event will be triggered when a message token with the same case id as the internal case token of the activity is received. When it is triggered, it consumes the internal case token and produces a case token with the same case id as consumed on the exception flow.

End event of type “message” (sending)



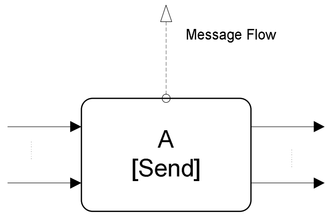
When an end event of type “message” has received all case tokens of a specific case, it will consume the internal case token of the (sub) process, it will produce a message token with the same case id on its outgoing message flow and it will produce one token with the same case id as the consumed one on all outgoing sequence flows of the (sub) process.

Task of type “receive” (receive)



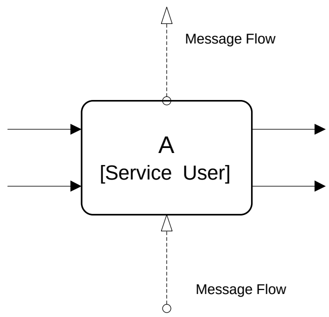
When a task of type “receive” has an internal token, it is waiting for a message token with the same case id as the internal case token. When such a token arrives, it consumes the internal case token and produces a case token for each outgoing sequence flow.

Task of type “send” (send)



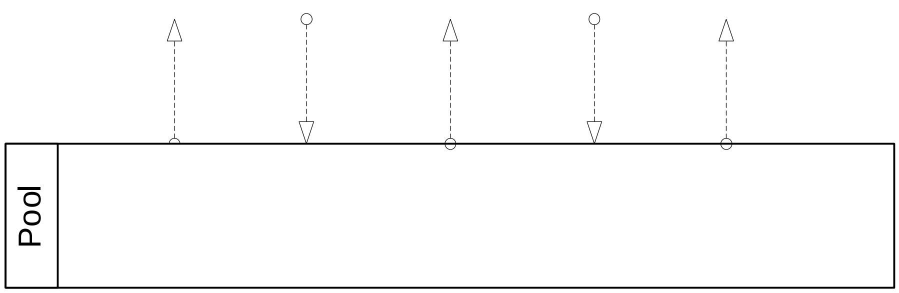
When a task of type “send” has an internal case token, it will consume the internal case token; it produces a message token with the same case as the consumed internal case token on its outgoing message flow and produces one case token with the same id for each outgoing sequence flow.

Task of type User or Service (send & receive)



When a task of type “service” or “user” receives a case token on one incoming sequence flow, it produces an internal case token, sends a message token with the same case id on its outgoing sequence flow and waits for a message token with the same case id to arrive. When such a message token arrives, it consumes the internal case token and produces a case token with the same case as consumed on each outgoing sequence flow.

Pool (send & receive)



An empty pool can be used to show a participant as a “black box”. It only shows the communication with the participant, but the activities responsible for the communication are not shown.

5. BPMN: the translation and implementation

For the translation we split the BPMN model into smaller pieces. A node with a variable number of incoming and outgoing sequence flows is called a pattern. For each pattern we must create a corresponding piece of PNML with the same semantics. When we translate each pattern and connect those translations, the translation of the whole process is completed.

5.1. Possible connections of an activity

Within a process, an activity can have incoming and outgoing message flows, incoming and outgoing sequence flows and when the activity is interruptible, it has an exception flow. For the readability of the resulting process, every activity will be converted into a PNML subnet. In that subnet there will be a transition that represents the task, or a subnet that represents the sub process. Every subnet will have some places that are used to translate the communication in the BPMN process. I will distinguish six kinds of places, each with their own color:

(white)	Normal place
(yellow)	Normal case sensitive place
(orange)	Case sensitive reference place
(red)	Case sensitive event place
(blue)	Case sensitive message place
(purple)	Case sensitive reset place

Multiple cases can concurrently exist in the BPMN process. In Petri nets the cases are represented by case tokens with different case ids. We need to use *case sensitive* places in the Petri nets. The incoming sequence flows and outgoing sequence flows are translated with case-sensitive places and directed arcs. Reference places within a subnet will be colored orange.

A task or event may be able to send and receive messages (message tokens) on message flows. This behavior will be handled by message places, which will be colored blue.

Another way an activity can be influenced is through an event. This behavior will be handled in the Petri net by an event place (colored red). An exception can be received from another activity (a case token) or from another process (a message token).

Finally, we need a construction to remove all tokens of a specific case, to support the BPMN terminate event. In each subnet there must be a transition which can reset all places. Because a subnet can contain other subnets, we need a reset place that can trigger the reset transitions for a child subnet. There must also be a reset place for triggering the parent net, because the terminate event can occur at any level of the process. See the translation of the terminate event in section [5.5.3.8].

In [Figure 25] below all possible connections of an activity are shown:

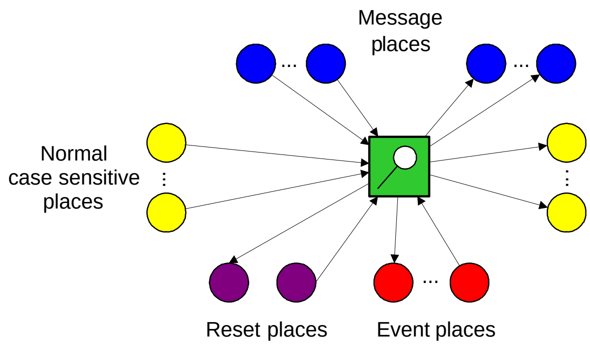


Figure 25: All possible connections of an activity

In the remaining part of this chapter the translation of all elements of the BPMN are enumerated.

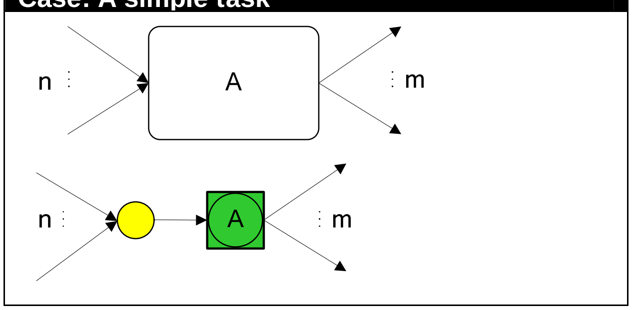
5.2. The translation of activities

Activities can have internal markers or can be interruptible. However, we start with the translation of a simple task – without internal markers or interrupt events – and a simple sub process.

5.2.1. Simple tasks and simple sub processes

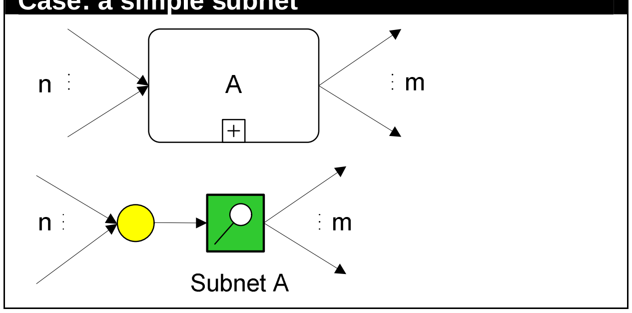
A simple task can easily be translated into a transition. Because each incoming sequence flow starts the task, the transition is preceded with a case sensitive place that receives all case tokens and is connected to the transition with a single arc.

Case: A simple task



A simple sub process can be translated with a subnet. Because each incoming sequence flow starts the sub process, the subnet is, just like the task, preceded with a case sensitive place that receives all case tokens and is connected to the subnet with a single arc. The content of sub process “A” will be translation of the content of the sub process “A”.

Case: a simple subnet



The translation of activities with internal markers or activities that are interruptible is shown in the following sections. The markers are translated one-by-one in a specific order. Often a marker adds an extra level of hierarchy to the sub process.

- If the activity is an interruptible activity, start with [Section 5.2.2]. otherwise:

- If the activity has an internal marker, start with [Section 5.2.3].

5.2.2. Interruptible activities

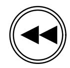
An interruptible activity, a task or a sub process, contains an interrupt event with an exception flow. There are six types of interrupt events:

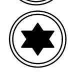

Interrupt event of type “message”


Interrupt event of type “rule“

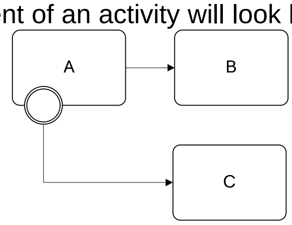

Interrupt event of type “error”


Interrupt event of type “timer “

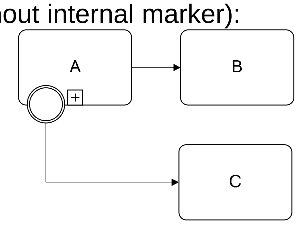

Interrupt event of type “compensation“


Interrupt event of type “multiple”

An interrupt event of an activity will look like (shown without internal marker):



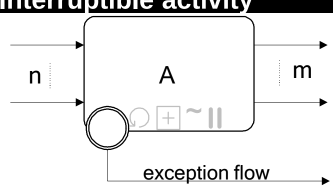
or

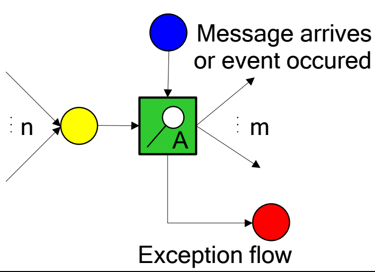


The interrupt event of an interruptible activity may receive a case token or message token from respectively another event or process. When the interrupt event of the interruptible activity is triggered during execution of the activity, the activity will not end normally, but will follow the exception flow.

The general scheme for an interruptible activity is:

Interruptible activity



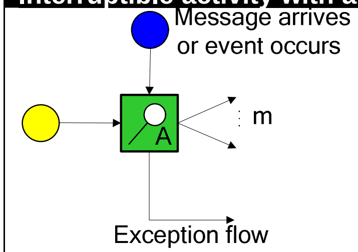


The content of subnet “A” will depend on the interrupt event of the activity and on the internal markers the activity can have.

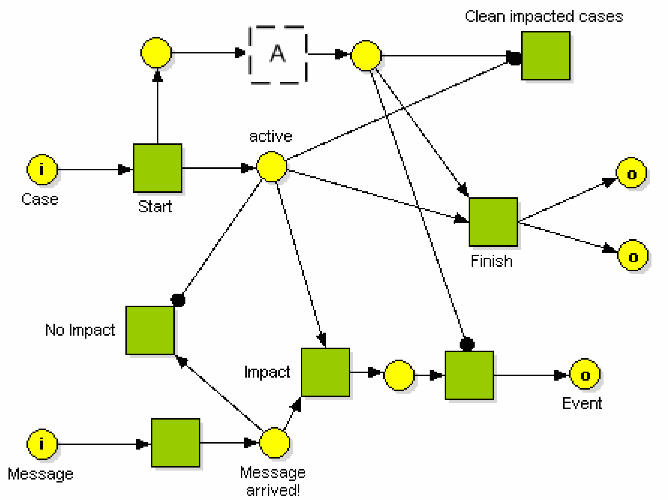
Depending on the internal markers, the translation will be different. The dotted rectangle in the translations below will be the translation of the activity itself including the internal markers. See the end of this section where the translation of that part can be found.

Interrupt activity of type "rule", "error", "compensation", "timer" or message

Interruptible activity with an interrupt event



With subnet "A":

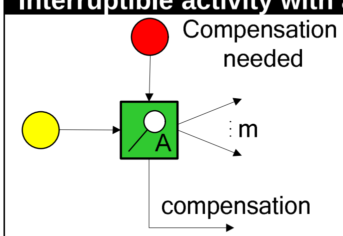


With the "active" place we can now check whether activity A is active or not and so determine the impact of the message. When activity A is active and a message arrives, the "Impact" transition will fire, it will remove the token in the place called "active" and it will put a token in its output place. When A is not active and a message arrives the "No impact" transition will clean up the token. The "Clean impacted cases" transition is needed to remove the tokens of cases that are interrupted.

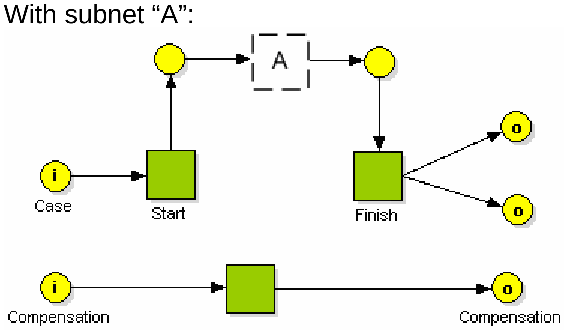
Interrupt activity of type "compensation"

When compensation is needed, a case token is received from an intermediate compensation event, resulting in the execution of the compensation activity C.

Interruptible activity with an interrupt event of type "compensation"



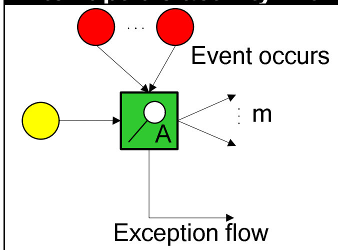
With subnet "A":



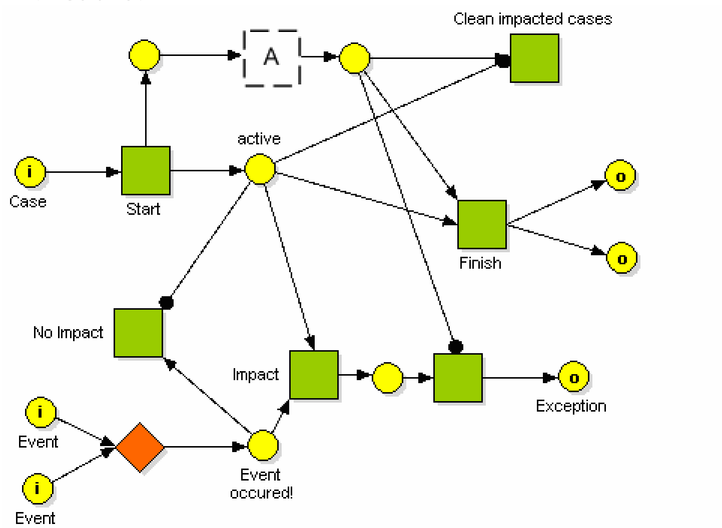
Multiple

The attributes of the interrupt event of type "multiple" determine which events can interrupt the activity. When one of the events occurs, the activity is interrupted.

Interruptible activity with an interrupt event of type "multiple"



With subnet "A":



The translation of the dotted rectangle "A" will be:

- If the activity has internal markers see [Section 5.2.3.] otherwise:
- If the activity is a sub process: a simple sub net
- If the activity is a task: a simple task

5.2.3. Internal Markers

Activities can have internal markers. In the table below the internal markers that tasks and sub processes can have are shown:

	Symbol	Internal Marker	Task	Sub process
1	⊞	Sub process Marker		•
2	↻	Standard Loop Marker	•	•
3	‖	Multi-Instance Loop Marker	•	•
4	◀◀	Compensation Marker	•	•
5	~	Ad-hoc Marker		•

See [Appendix C] for an overview of the combinations of internal markers a task or sub process can have.

The translation of the markers must be done in the following order:

(1) Compensation marker
(2) Standard or multi-instance loop marker

(3) Ad hoc marker
(4) Sub process marker

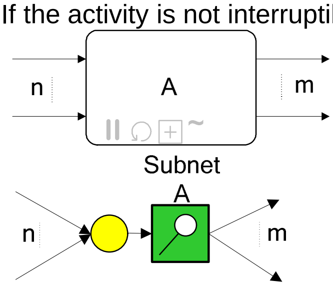
- If the activity has a compensation marker see [Section 5.2.5.], otherwise
- For the other markers see [Section 5.2.4.]

5.2.4. Add an extra level of hierarchy

An activity without a compensation marker and with *n* incoming sequence flows and *m* outgoing sequence flows can be translated into:

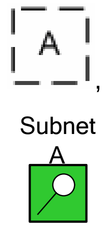
Add extra level

If the activity is not interruptible:



Subnet “A” will contain the translation of the activity including all internal markers.

If the activity is interruptible, the dotted rectangle “A” in the translation will be a subnet:



Subnet “A” will contain the translation of the activity including all other internal markers.

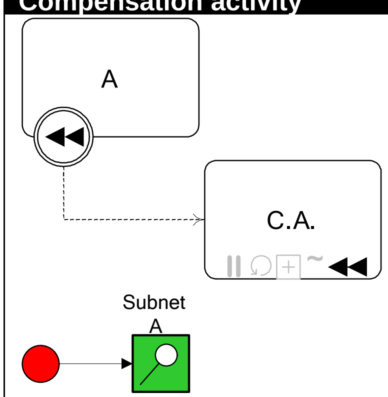
The content of “Subnet A” is the translation of the activity, which depends on the other markers:

• If the activity has a standard loop marker:	[Section 5.2.6.]	otherwise:
• If the activity has a multi-instance loop marker:	[Section 5.2.7.]	otherwise:
• If the activity has an ad hoc marker:	[Section 5.2.8.]	otherwise:
• If the activity has only a sub process marker:	a simple subnet	otherwise:
• If the activity has no markers:	a simple task	

5.2.5. Compensation Activity

If the activity is a compensation activity, it has no outgoing sequence flows. The compensation activity has **one** incoming sequence flow that is connected to an interruptible activity with an interrupt event of type “compensation”. When the interrupt event is triggered, this compensation activity receives a case token.

Compensation activity



The subnet will contain the translation of the activity with the remaining markers.

The content of “Subnet A” is the translation will be:

• If the activity has a standard loop marker:	see [Section 5.2.6.]	otherwise:
• If the activity has a multi-instance loop marker:	see [Section 5.2.7.]	otherwise:
• If the activity has an ad hoc marker:	see [Section 5.2.8.]	otherwise:

- If the activity has only a sub process marker: a simple subnet
- If the activity has no markers: a simple task

5.2.6. Standard loop marker

The standard loop has a Boolean expression (*LoopCondition* attribute) that will be evaluated before or after each loop (depending on the *TestTime* attribute). When the expression evaluates to true, the loop is executed one more time.

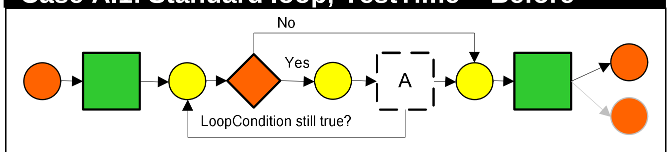
The following attributes determine the behaviour of the standard loop:

Attribute	Possible values
Activity.TestTime	Before | After
Activity.LoopCondition	Expression which resolves to a Boolean

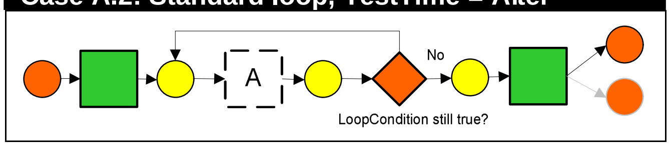
The translation of different standard loops is shown in the following cases:

Case	Activity. TestTime
A.1	Before
A.2	After

Case A.1: Standard loop, TestTime = Before



Case A.2: Standard loop, TestTime = After



In the above translations, the dotted rectangle "A" will be:

- If the activity has a an hoc marker: see [Section 5.2.8.], otherwise
- If the activity is a sub process: a simple subnet, otherwise
- If the activity is a task: a simple task

5.2.7. Multi-Instance loop marker

The multi-instance loop has a loop expression stored in the *MI_Condition* attribute. The expression will be evaluated once and determines the number of times the activity will be repeated. The execution of the activity can be done sequential or in parallel, depending on the *MI_Ordering* attribute. When the execution is done in parallel, the *MI_FlowCondition* attribute determines when and how many output tokens are produced. The attributes that determine the behaviour of the multi-instance loop are:

Attribute	Possible values
Activity.MI_Condition	Expression → Integer
Activity.MI_Ordering	Sequential | Parallel
[Parallel] Activity.MI_FlowCondition	None | One | All | Complex

The translation of different loop types is shown in the following cases:

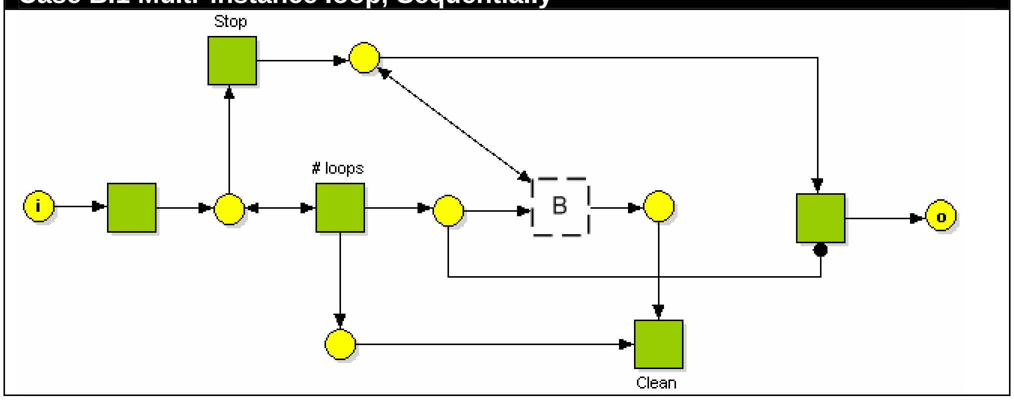
Case	Activity.MI_Ordering	Activity.MI_FlowCondition
B1	Sequential	
B2.1	Parallel	None
B2.2	Parallel	One
B2.3	Parallel	All

B2.4	Parallel	Complex

In the translations below there are as many case tokens created as loops that must be executed. The timestamp of the created case tokens is the same, because the production of tokens is done with a transition without processing time. The case tokens are consumed by a timed transition. Because the timestamps of the case tokens are the same and the consumption of the tokens does not cost time, the loops are executed in parallel.

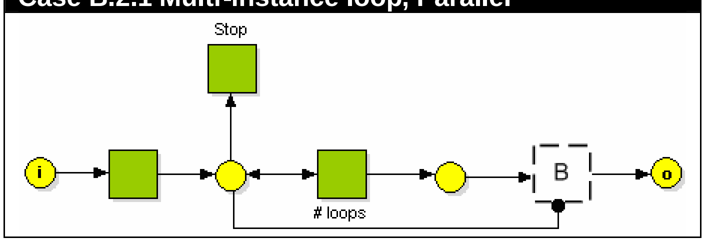
Case B1: Activity.MI_Ordering: Sequentially
The loops are executed one by one. After all loops have been completed, the output is created.

Case B.1 Multi-instance loop, Sequentially



Case B2.1: Activity.MI_Ordering: Parallel & Activity.MI_FlowCondition: None
Every loop creates an output token.

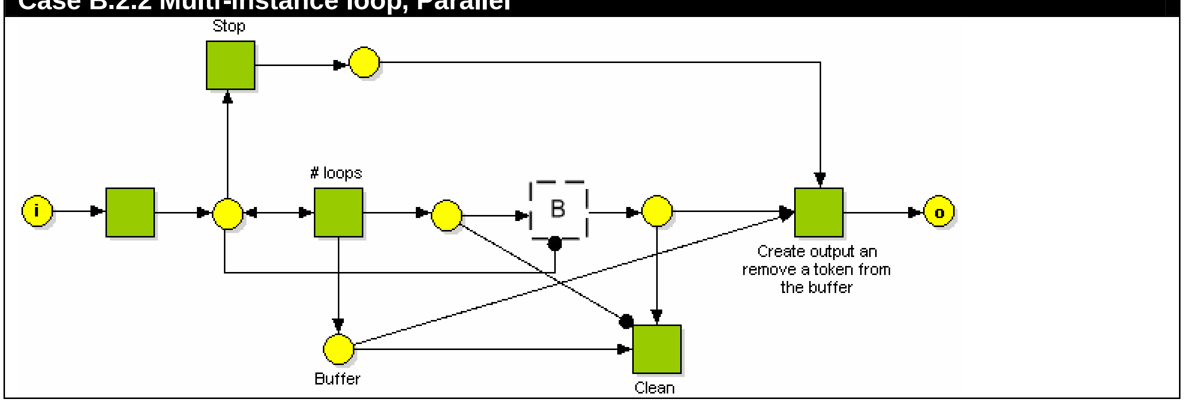
Case B.2.1 Multi-instance loop, Parallel



* The dotted rectangle “B is or contains a timed transition.

Case B2.2: Activity.MI_Ordering: Parallel & Activity.MI_FlowCondition: One
Only the first loop creates an output token, the rest of the loops are executed, but will not produce an output token.

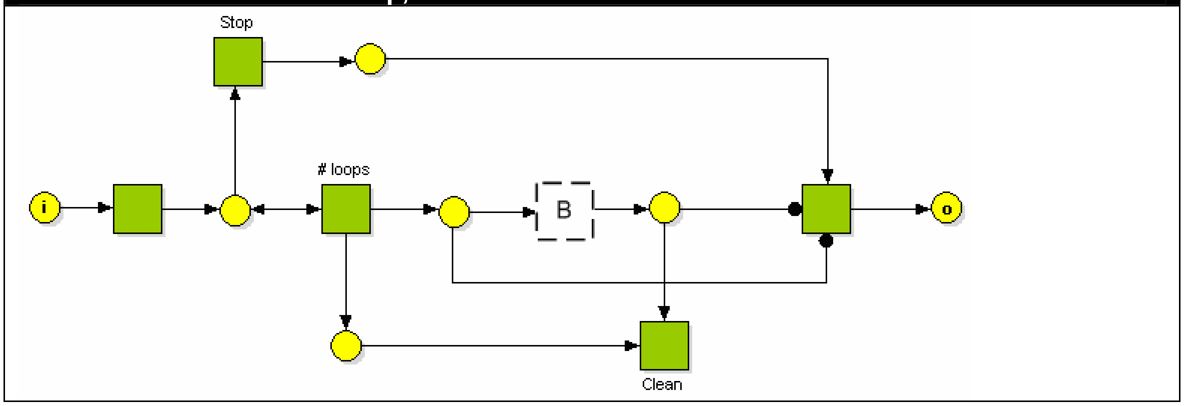
Case B.2.2 Multi-instance loop, Parallel



* The dotted rectangle “B is or contains a timed transition.

<u>Case B2.3: Activity.MI_Ordering: Parallel & Activity.MI_FlowCondition: All</u>
When all loops are completed one token will be produced as output.

Case B.2.3 Multi-instance loop, Parellel



* The dotted rectangle “B is or contains a timed transition.

<u>Case B2.4: Activity.MI_FlowCondition: Complex</u>
We don’t translate this type of multi-instance loop, because the output behavior is not translatable into a Yasper model.

In the above translations, the dotted rectangle “B” will be:

- If the activity has a an hoc marker: see [Section 5.2.8], otherwise
- If the activity is a sub process: a simple subnet, otherwise
- If the activity is a task: a simple transition

5.2.8. Ad hoc

Only sub processes can contain an ad hoc marker. The translation of the ad hoc marker includes the translation of the sub process marker. An adhoc sub process contains only activities, which are not connected with sequence flows. This means that the activities may be executed in any order and as many times as long as the expression in the *adhocCompletionCondition* attribute is false. The attributes that determine the behaviour of an ad-hoc process are:

Attribute	Possible Values
AdHocOrdering	Parallel | Sequential
AdHocCompletionCondition (0 | 1)	Expression

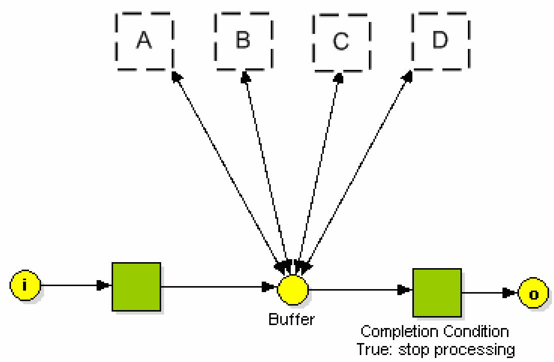
The translation of different ad hoc activities is shown in the following cases:

Case	**Activity. AdHocOrdering**
A	Sequential
B	Parallel

Case A: AdHocOrdering: Sequential

The dotted rectangle will be a subnet with the following content:

Case A: AdHocOrdering: Sequential

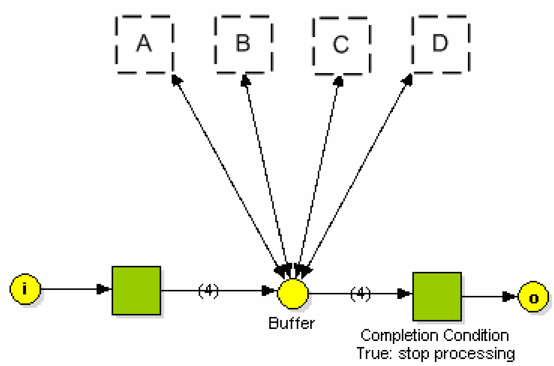


The activities can be executed (one at a time) as there is a token in the “buffer” place. If the “Completion Condition True: stop processing” transition fires, the token is removed and an output token is produced.

The activities (A, B, C or D) are translated as described in this chapter.

The dotted rectangle will be a subnet with the following content:

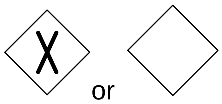
Case B: AdHocOrdering: Parallel

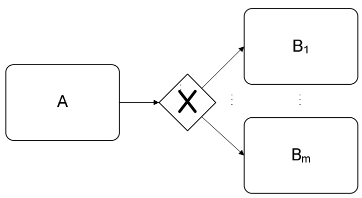


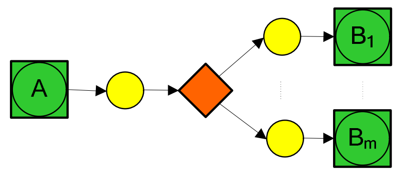
The first transition will create as many token in his output place as there are tasks in the process. The tasks can be executed until there are tokens in the “buffer” place. If the “Completion Condition True: stop processing” transition fires, all tokens are removed and an output token is produced.

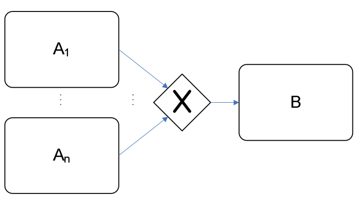
5.3. Translation of gateways

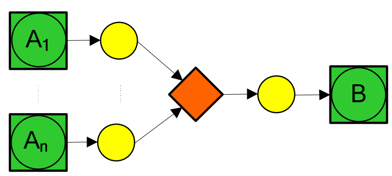
5.3.1. Exclusive gateway (Data Based)



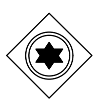




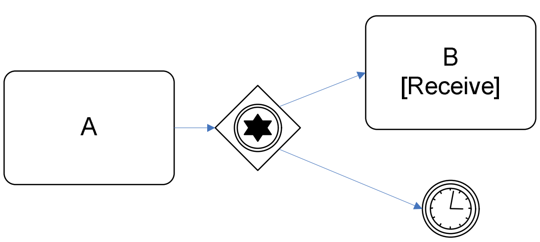


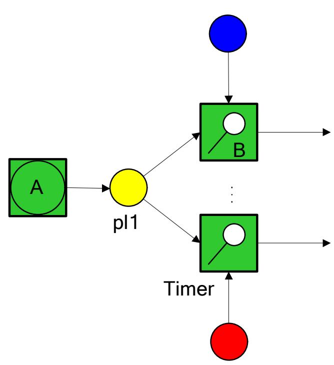


5.3.2. Exclusive gateway (Event Based)



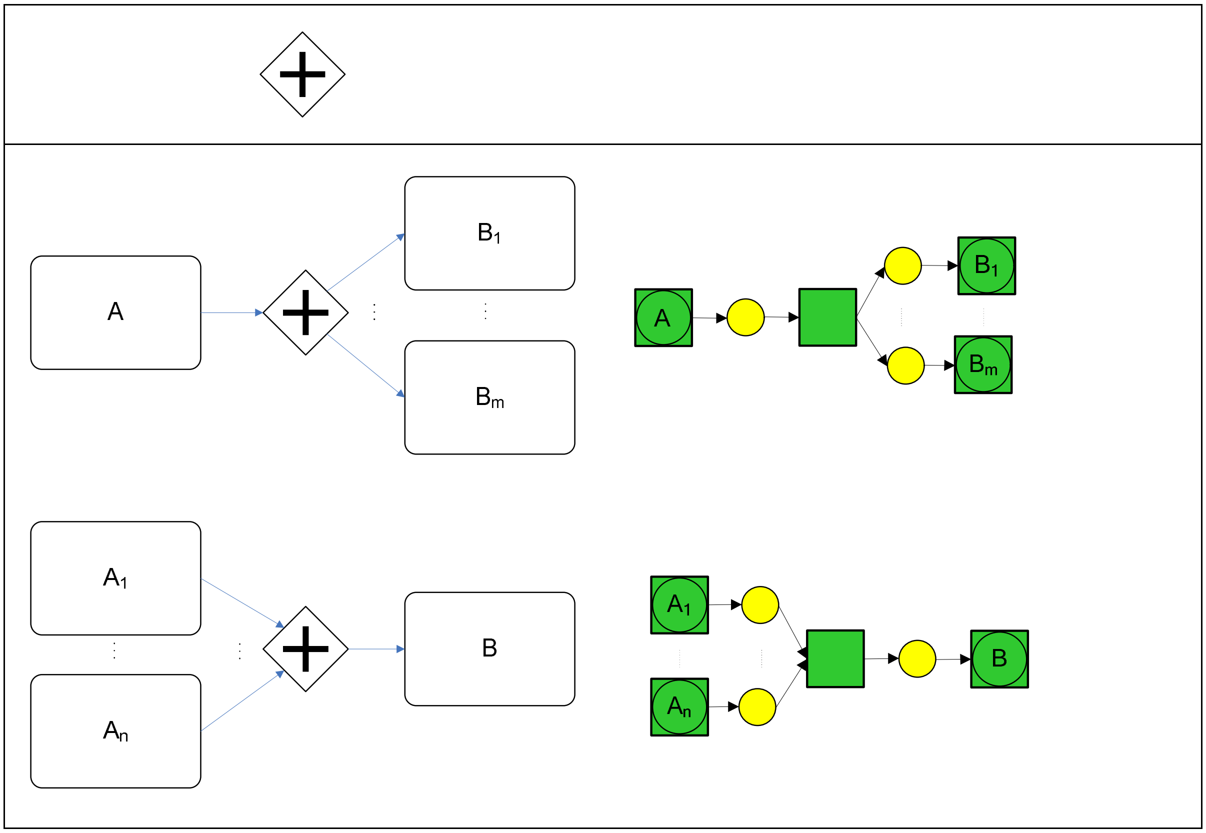
The outgoing sequence flows of an exclusive gateway must be connected with intermediate events or with tasks of type “receive”.





The events will share the input place pl1. The first event that is triggered will consume the token in pl1.

5.3.3. Parallel gateway



5.3.4. Inclusive gateway

We do not support inclusive gateways. The behavior of this type of gateway is not translatable in a Yasper model. The inclusive gateway uses global data in its expression. Yasper can only use data that is locally available.

5.3.5. Complex gateway

We do not support complex gateways. Use a net of simple gateways instead.

5.4. The translation of events

Events can occur in two different ways in a process. An event in normal flow works as a delay mechanism or as a mechanism to throw errors by sending case or message tokens to other events. An interruptible activity is a mechanism to catch errors. When an error occurs, the activity will not end normally, but the exception flow will be followed.

5.4.1. Events in normal flow

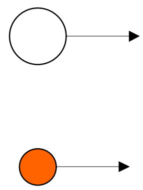
5.4.1.1. Translation of start events

In the next section a translation is given for each type.

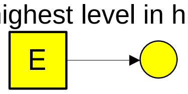
None

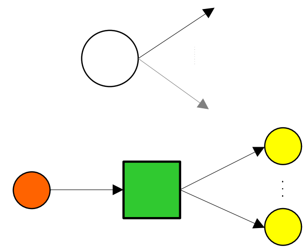
A start event will be the beginning of a sub process and is translated by a reference place in the Petri net. Because a start event will generate a token on all its outgoing sequence flows, we must translate a start event with an and-split in the following way:

1 outgoing sequence flow: Multiple outgoing sequence flows:

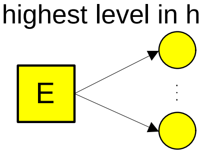


On the highest level in hierarchy



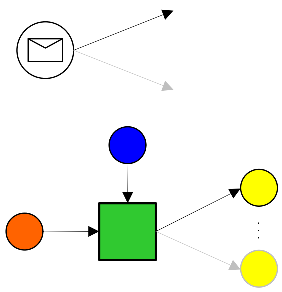


On the highest level in hierarchy



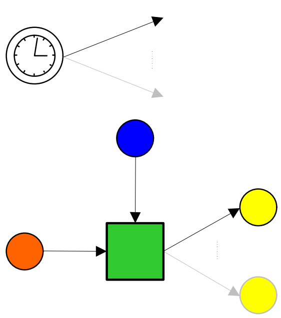
Message

The process is waiting for a message and will start when a message token arrives from a participant. When the message token arrives, it will consume the message token and the internal case token and produce a case token with the same case id as the consumed one(s) on all outgoing sequence flows.



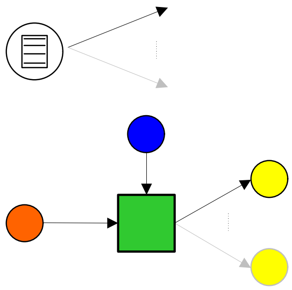
Timer

We handle a start event of type "timer" as a transition that receives a message token from a separate process. That transition is waiting for that message token to arrive. When the message token arrives, it will consume the internal case token and the message token and it will produce a case token with the same id as the consumed token on all outgoing sequence flows.



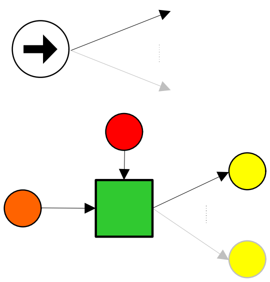
Rule

We handle a start event of type "rule" as a transition that receives a message token from a separate process. That transition is waiting for that message token to arrive. When the message token arrives, it will consume the internal case token as well as the message token and produce a case token with the same id as the consumed token on all outgoing sequence flows.



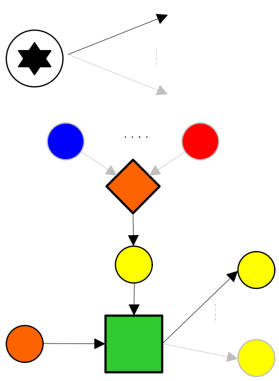
Link

This start event is triggered when a case token is received from a link end event within the same parent process. When the case token is received, it will consume the case token and the internal case token and produce a case token with the same case id on all outgoing sequence flows.



Multiple

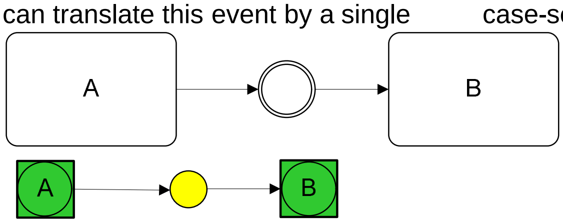
The attributes of a multiple start event will determine which events may trigger this event. Only one of them is sufficient to trigger the event. If the start event is triggered, it consumes the internal case token and produces a case token with the same case id as the consumed one on all outgoing sequence flows.



5.4.2. Intermediate events

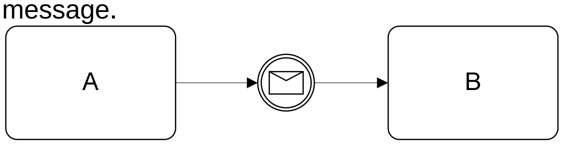
None

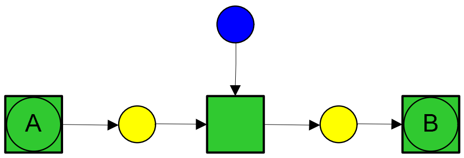
No type is set for the intermediate event. It is a point in the process where the state changes. We can translate this event by a single case-sensitive place.



Message

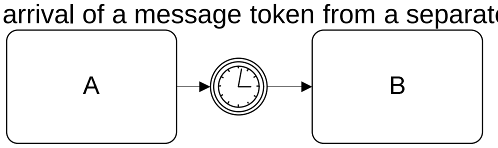
The process is waiting for a message token to arrive from another process. We assume that the case id of the message token has the same case id as the process that is waiting for the message.

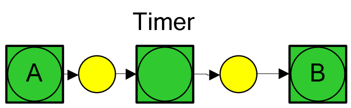




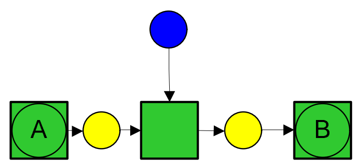
Timer

An intermediate timer event can be translated with a timed transition if the process is waiting for a fixed amount of time, e.g. wait 4 hours. If the *TimeCycle* attribute is set, or a variable amount of waiting time, e.g. each hour or Monday 09:00, we translate the intermediate timer event with the arrival of a message token from a separate time process.





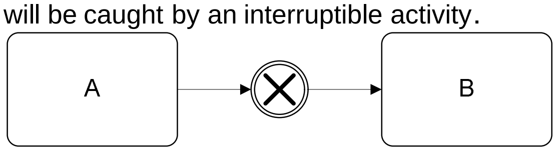
Fixed amount of time

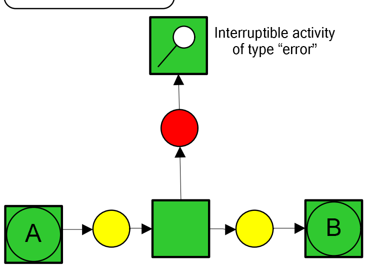


A specific time-date or cycle

Error

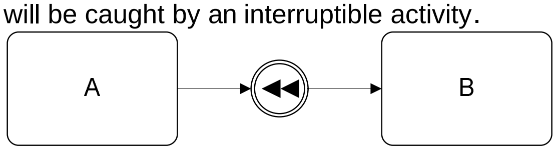
This event will throw an error, by producing a case token on an outgoing sequence flow, which will be caught by an interruptible activity.

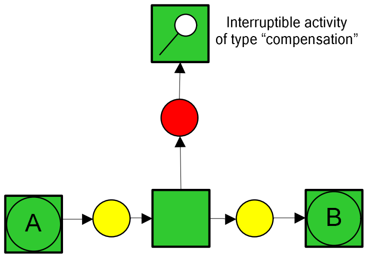




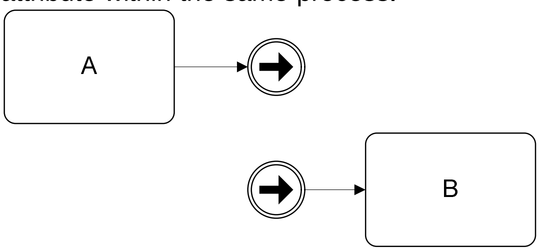
Compensation

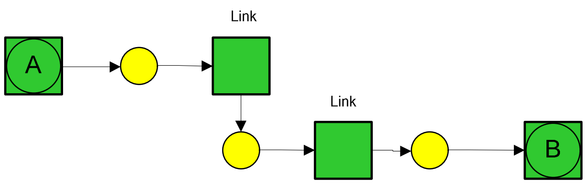
This event will throw an exception, by sending a case token on an outgoing sequence flow, which will be caught by an interruptible activity.





Link
An Intermediate Link Event will trigger another intermediate link event with the same *LinkId* attribute within the same process.

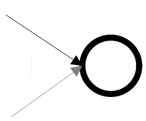


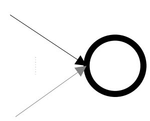


5.4.3. Translation of end events

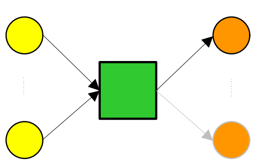
In this section the translation for each type of end event will be given.

None
When all case token of a case are consumed, the end event will produce a case token with the same case id on each outgoing sequence flow of the sub process it is contained in. When the end event is the end event of the top level process, the end event is translated with a collector. An end event can have multiple incoming sequence flows. We will require that the incoming sequence flows are all parallel.

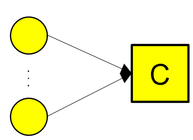




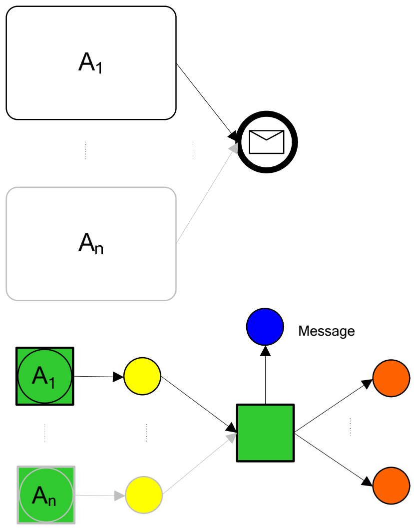
End event in a subnet:



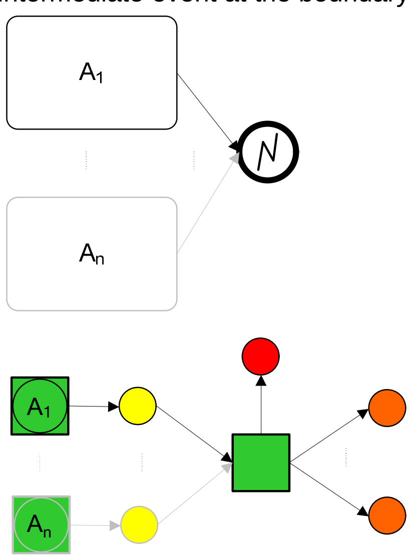
End event in the top level process:



Message
When all tokens of a case are consumed, a message token with the same id is produced on its outgoing message flow and a case token with the same id is produced on all outgoing sequence flows of the sub process it belongs to.

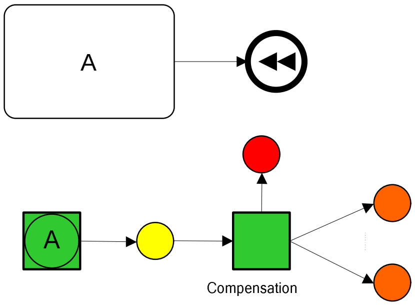


Error
When all tokens of a case are consumed, an error, represented by a case token with the same id as the consumed ones, is generated and a case token with the same id is produced on all outgoing sequence flows of the sub process it belongs to. The produced error will trigger an intermediate event at the boundary of an activity.



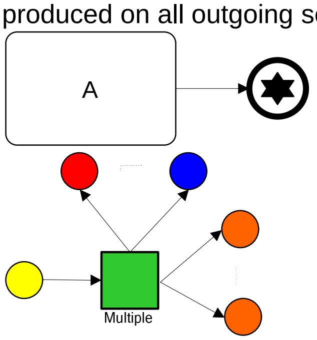
Cancel
This end event may only be used in a transaction process. We do not support transactions.

Compensation
When all tokens of a case are consumed, a case token with the same id as the consumed ones, is generated and a case token with the same id is produced on all outgoing sequence flows of the sub process it belongs to. The produced case token will trigger an intermediate compensation event at the boundary of an activity.



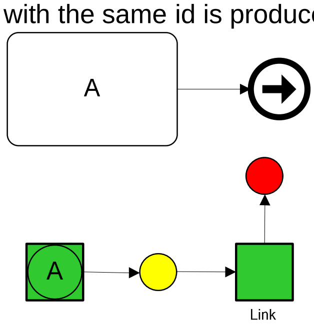
Multiple

When all tokens of a case are consumed, multiple tokens with the same id as the consumed ones are produced, depending on the attributes of the event. A case token with the same id is produced on all outgoing sequence flows of the sub process it belongs to.



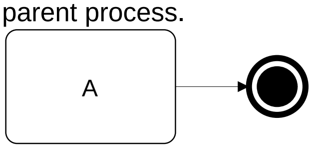
Link

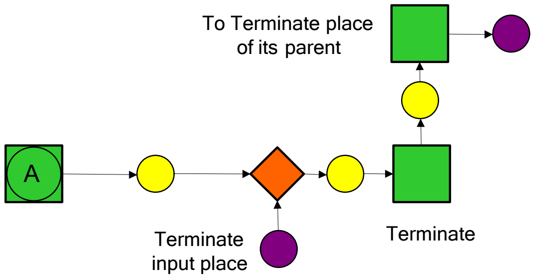
When all tokens of a case are consumed, a case token with the same id as the consumed ones is produced that will trigger a start event of type link within the same parent process. A case token with the same id is produced on all outgoing sequence flows of the sub process it belongs to.



Terminate

If a process contains a terminate event, all sub processes must be equipped with a terminate transition that removes all the tokens of a specific case from all places in the process and send a “terminate” token, represented by a case token, to the terminate transitions of its sub processes. When all tokens of a case are consumed by a terminate end event, this end event terminates the processing of the whole process for the consumed case. It sends a case token to the terminate transition of its own process, to its own sub processes and also to the terminate transition of its parent process.





The terminate transition must consume all case tokens with the same case as the consumed one(s) on this process level. It will send a ”terminate” token to its parent process and to all its sub processes. The reset arcs to all the places of the subnet are not shown in the figure aside.

5.5. Correctness of the translation

The informal semantics introduced in [Section 3.4] describe a transition system. The correctness of the translation of a process can be formalized with the notion of a bisimulation relation between the two models, see [Baeten,Weijland, 1990]. Then it is in principle possible to verify if such bisimulation relation exists for a given start state (this is always the same one). A complete formal proof goes beyond the scope of this project.

5.6. Implementation

The implementation of the conversion between BPMN and PNML was a part of this master's thesis project. The BPMN processes were made using a software application, called Deloitte Process Modeler (DPM), which was under development at Deloitte. Within that application, not all elements of the BPMN standard were supported yet, but enough to create most of the processes. The Deloitte Process Modeler stores a process using XML technology. During my graduation period, there was no XML standard for saving BPMN processes; such a standard was still in development by the BPMI.
During the research, some of the elements appeared to be hard to convert into PNML. We made some requirements for the BPMN process:

- REQ.1 The OR-Gateway may not be used. *When this gateway is used for joining flow it uses information about which paths were taken at runtime at a downstream object. In a specific case we can translate this behavior, but in general this information is not available in Petri nets.*
- REQ.2 The Complex-Gateway may not be used. *The split and join behavior of this gateway is stored in an expression. This gateway can be replaced by a network of simple gateways, exclusive and parallel gateways with the same behavior.*
- REQ.3 Every parallel path must eventually be joined with a parallel gateway. *This preserves the well-handled properties of the process.*
- REQ.4 Every alternative path must eventually be joined with an exclusive gateway. *This also preserves the well-handled properties of the process.*
- REQ.5 A gateway can be used for splitting and / or joining the flows. *Although gateways can split and join incoming and outgoing paths simultaneously, we only use one of those functions per instance.*
- REQ.6 Every process (level) must have one – and only one – start event. *A process is easier to translate when there is only one start event. A process with more than one start event can be remodeled to a process with exactly one start event.*
- REQ.7 Every sub process must have only one end event. *REQ.4 and REQ.5 ensure this requirement is met. A process is easier to translate when there is only one end event. A process with multiple end events can be remodeled to a process with one end event.*
- REQ.8 Transactions are not supported. *A transaction is handled by a transaction protocol.*

The actual translation of a BPMN process into a Petri nets file can be achieved in different ways, see [Figure 26].

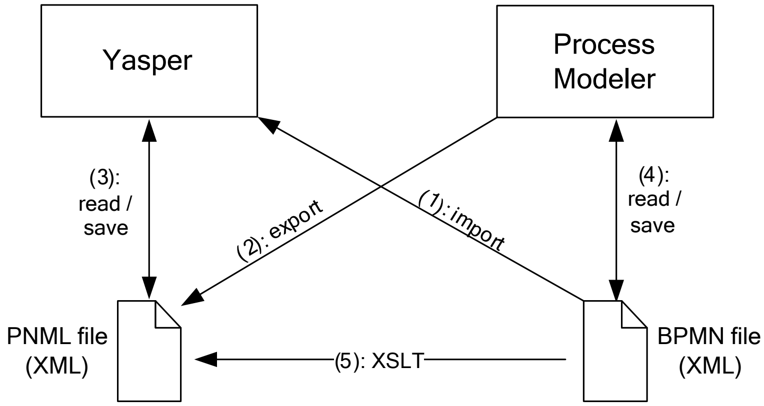


Figure 26: Conversion options

We could extend the Deloitte Process Modeler to support an export function to PNML (1). The other way round we could extend Yasper with an import function to support the BPMN file format (2). However, we have chosen for a more elegant solution (5). Because both file formats are based on XML, we could convert one format into the other with XSLT[4] [XSLT].
XSLT is a language for transforming XML documents into other XML documents. A transformation expressed in XSLT is a well-formed[5] XML document and called a transformation or a stylesheet in XSL formatting vocabulary. A XSLT file describes the rules for transforming the source tree into a result tree. The source tree is separated from the result tree. The structure of the result tree can be completely different from the source tree; the elements from the source tree can be filtered, reordered and arbitrary structures can be added.

A stylesheet contains a set of templates. A template has two parts: a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree. When a template is instantiated, it is always instantiated with respect to the current node and current node list. The current node is always member of the current node list. Many operations in XSLT are relative to the current node. XSLT uses XPath [XPath] for selecting elements for processing, for conditional processing and for generating text.
Schematically, the conversion will be performed as in the [Figure 27] below:

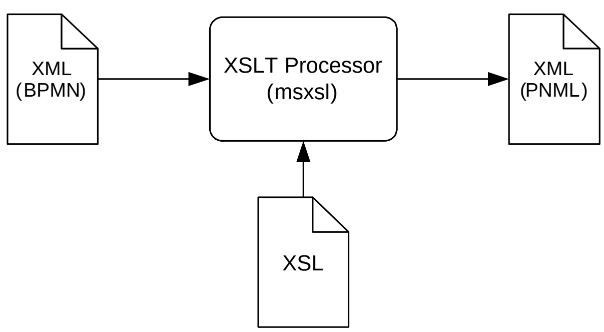


Figure 27: The conversion

The actual conversion is done by an XSLT processor. We have used the Microsoft XSLT Processor (msxsl.exe) to convert the BPMN file into a PNML file. The Microsoft XSLT processor

4 Extensible Stylesheet Language Transformations
5 An xml file is well-formed when it has exactly one root element and that all tags are nested properly. For an exact definition see: http://www.w3.org/TR/REC-xml/#sec-well-formed

is a command line tool, which takes some attributes. The command below will convert `input.xml` according to `conversion.xsl` into `output.xml`.

```
msxsl.exe input.xml conversion.xsl –o output.xml
```

The XSLT processor contains an xml parser, which will start reading the source file at the top node with a deep-first strategy. For each node the processor checks if the node is matched by one of the templates. If so, the template is instantiated and the result tree may be extended. When the source tree is processed completely the result tree is the final result of the transformation.

For the transformation of a BPMN file to a PNML file three XSLT files were created:

<u>BPMNElements.xsl</u>
This file contains the templates that match the language constructs in the BPMN file: tasks, sub processes, gateways, events and sequence flows. Each template collects information such as the *id*, the *graphical information* and the *name* of the element. The *type* of a BPMN element is checked with conditional statements and depending on the type the right PNML templates from the PNMLElements.xsl file are called. The *id* of the element is used to call a template which search for incoming and outgoing sequence flows.

<u>PNMLElements.xsl</u>
This file contains for each PNML element a template which can be used in the templates in the file above. Those templates can be called with parameters which are used to adjust e.g. positional information, names or ids.

<u>Aux_Functions.xsl</u>
Contains auxiliary functions used in the other files.

Below I will give a simplified example of the code transformation of a parallel gateway:

BPMN snippet:
```
<?xml version=”1.0” encoding=”UTF-8”?>
...
<Gateways>
  <Gateway id=”gw01”>
    <Name>Gateway1</Name>
    <GatewayType>ParallelAND</GatewayType>
  </Gateway>
<Gateways>
...
```

The snippet above contains the code for a simplified parallel gateway. Below the parts of the XSLT files that are used for the conversion are given:

BPMNElements.xsl snippet:
```
...
<xsl:template name=”Gateways”
match=”Gateways/Gateway”>
  <xsl:variable name=”name” select=”name/text”/>
  <xsl:variable name=”id” select=”@id”/>
  <xsl:call-template name=”transition”>
    <xsl:with-param name=”id” select=”$id”/>
    <xsl:with-param name=”name” select=”$name”/>
  </xsl:call-template>
</xsl:template>
```

```
...
```

PNMLElements.xsl snippet:

```
...
<xsl:template name="transition">
  <xsl:param name="id"/>
  <xsl:param name="name"/>
  <xsl:element name="transition">
    <xsl:attribute name="id" select="$id"/>
    <xsl:element name="name">
      <xsl:element name="text">
        <xsl:value-of select="$name"/>
      </xsl:element>
    </xsl:element>
  </xsl:element>
</xsl:template>
```

The XML parser will start reading the BPMN file. The *<Gateways><Gateway>*-tag will match the template with the name “Gateways” in the BPMNElements.xsl snipet. That template will store the *name* and the *id* of the gateway in two variables: *name* and *id*. Next, it will call the template with the name “transition” from the PNMLElements.xsl file. The template is called with two parameters, passing the *name* and the *id* of the gateway to the ”transition” template. Finally, the “transition” template reads the parameters and creates the corresponding code. The output will be:

```
<transition id="gw01">
  <name>
    <text>Gateway1</text>
  </name>
</transition>
```

<u>A visual example:</u>
In the [Figures 28 & 29] below, a BPMN process with its translation is given. A rectangle around a BPMN language construct has roughly the same position as the translation of the language construct in the resulting Petri net.

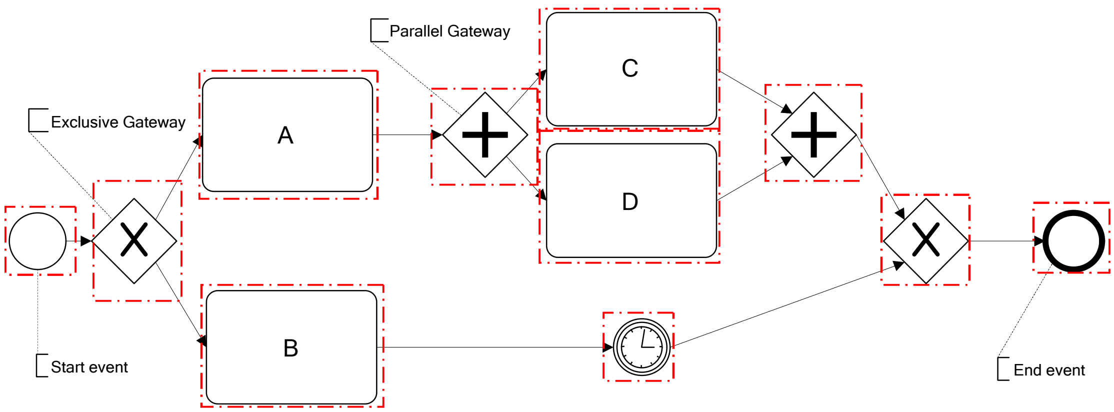


Figure 28: The BPMN process to translate

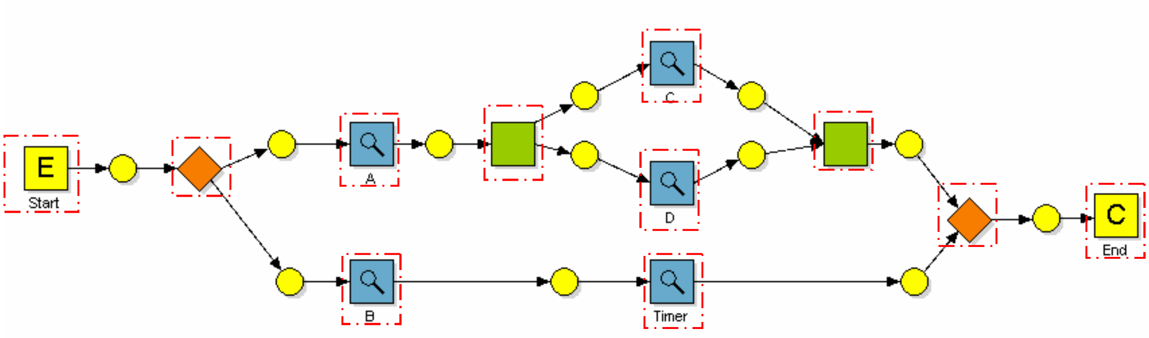


Figure 29: The resulting Petri net

The compact disc on the last page of this paper contains the XSLT files and the XSTL processor msxsl.exe. It also contains an example of a translation of a BPMN process.

5.6.1. An extension for verification

The main goal for translating a BPMN process into a Petri net was to be able to analyze the translation of the process in order to make statements about the original process. With a small addition to the templates, we can even trace back the element or elements that introduce the error in the original process. For each PNML element created in the stylesheet, we need to add information about which BPMN element is responsible for the creation of the PNML element. This information can be used to trace back the error to the elements in the original process when an error is found in the translation.

6. Conclusion

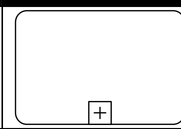
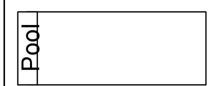
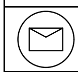
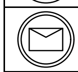
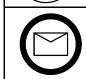
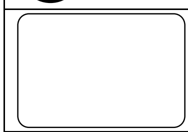
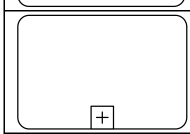
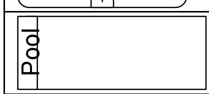
- BPMN is a compromise from a group that has self-concern in BPMN to incorporate features from their own techniques. The consequence is that BPMN is complex and contains constructs with unclear semantics.
- The lack of semantics of BPMN was the reason for us to define one ourselves. This, informal described, semantics covers most of BPMN and shows that there are several unclear constructs in BPMN.
- The presented informal semantics of BPMN are derived from the official specification. The semantics describe a transition system based on Petri nets for the BPMN language constructs.
- The C++ application for the conversion of a Provision process model can only be used when the user has access to Provision Enterprise. The provision language constructs are translated into PNML as described in [Chapter 2].
- The translation of BPMN models is platform independent, as long as a XSLT processor for the platform is available.
- For the implementation of the translation of BPMN models into Petri nets, the Process Modeler has been used to create process models in the BPMN standard. Because the software was in development, not all language constructs were implemented yet. However, the constructs that were implemented are translated as described in [Chapter 5].

7. Recommendations

- When using BPMN, use only well defined constructions: omit the use of the inclusive gateway and other elements without a well defined behavior.
- The XSLT templates can be extend when all language constructs are implemented in the Deloitte Process Modeler. With the translations on the conceptual level, the templates for the missing language constructs can be created.

Appendix A: Objects allowed to send or receive message flows

The table below shows the objects that can be connected to each other through a message flow. The ● symbol indicates that the object in the row can be connected to the object listed in the column. Message flows can only connect objects which are contained in different pools. Only tasks where the *TaskType* attribute is set to “receive” can be connected with a message flow. See [Section 2.2.1] for more information.

From / To						
						
	●	●		●	●	●
						
	●	●		●	●	●
	●	●		●	●	●
	●	●		●	●	●
	●	●		●	●	●

Appendix B: The different events types

In this appendix an overview is given of the places where the different event types can occur.

There are 9 types of intermediate events. In the table below the place where each type of intermediate event may occur is listed:

Intermediate events	Normal Flow	Boundary
None	OK	
Message	OK	OK
Timer	OK	OK
Error	OK	OK
Cancel		OK* transactions only
Link	OK	
Compensation	OK	OK
Rule		OK
Multiple		OK

There are 6 types of start events:

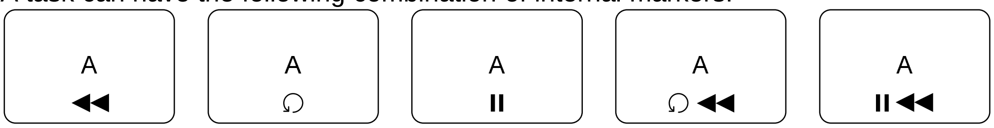
Start events
None
Message
Timer
Rule
Link
Multiple

There are 8 types of end event:

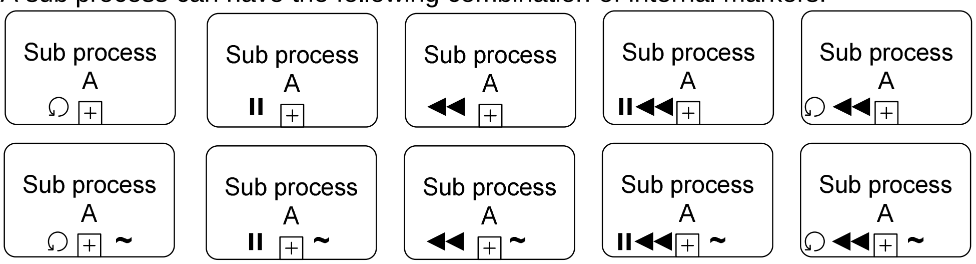
End Events
None
Message
Error
Cancel
Compensation
Link
Terminate
Multiple

Appendix C: Internal markers of an activity

A task can have the following combination of internal markers:



A sub process can have the following combination of internal markers:



Appendix D: Unfold a sub process

To successfully expand a sub process it must meet the following requirements:
(1) The start event has one outgoing sequence flow
(2) The end event has one incoming sequence flow

When the process does not meet requirement (1), the process can be remodeled to meet the requirement. The following replacements must be made:
- The start event must be replaced by a start event with one outgoing sequence flow connected to a parallel gateway.
- The targets of the outgoing sequence flows of the original start event become the targets of the outgoing sequence flows of the parallel gateway.

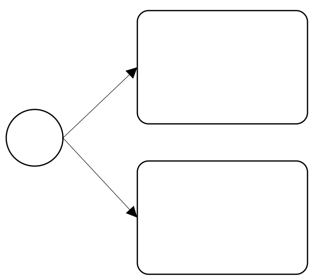


Figure 30: before replacement

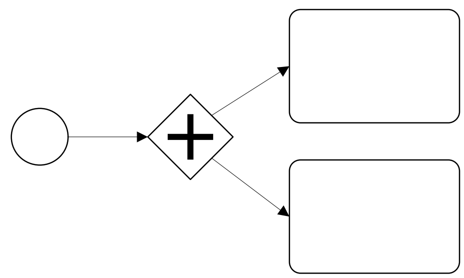


Figure 31: after replacement

The part of a process in [Figure 30] will be replaced with respect with rule (1) with the part in [Figure 31].

When the process does not meet requirement (2), the process must be remodeled to meet the requirement. The following replacements must be made:
- The end event must be preceded with a parallel gateway.
- The sources of the incoming sequences flows of the end event must be connected to the parallel gateway.[6]

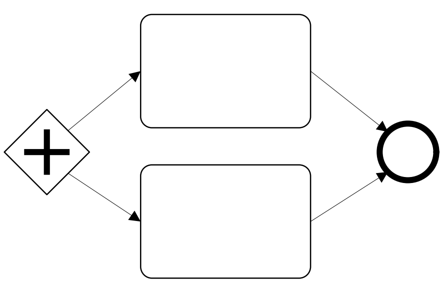


Figure 32: before replacement

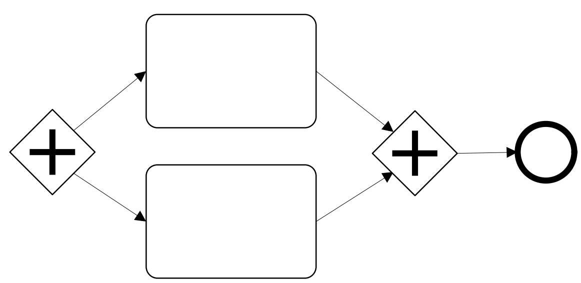


Figure 33: after replacement

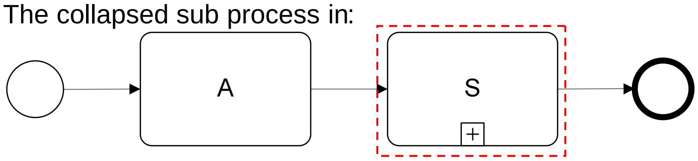
The part of a process in [Figure 32] will be replaced with respect to rule (2) with the part in [Figure 33].

When the sub process meets requirements (1) and (2), the start and end event can be removed from the subnet. The incoming connections with the subnet in the original process can be

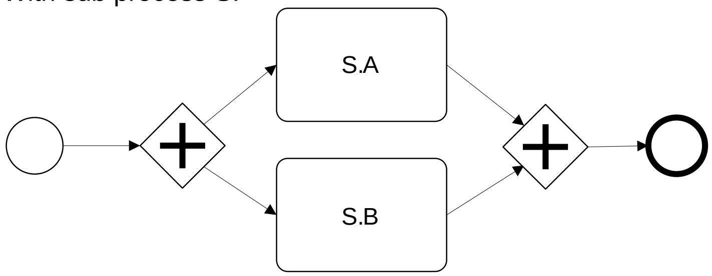
[6] This works only if alternative paths paired up with an exclusive gateway

connected with the first object in the unfolded sub process and the outgoing sequence flow of the original sub process can be connected to the last object.

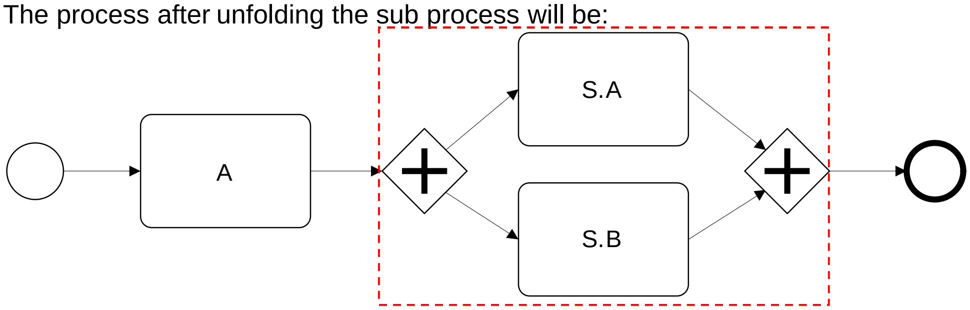
The collapsed sub process in:



With sub process S:



The process after unfolding the sub process will be:



A. References

[J.L. Peterson, 1981] J.L. Peterson: *Petri Net Theory and the Modeling of Systems* Prentice Hall, Englewood Cliffs, 1981.

[W. Reisig, 1985] W. Reisig: *Petri nets: An introduction* Springer-Verlag, New York, 1985.

[K. Jensen, 1981] K. Jensen: *Colored Petri nets and the invariant method* TCS, Aarhus University, Computer Science Department, 1981.

[R. Valette, 1981] R. Valette: *Analysis of Petri Nets by Stepwise Refinement* JCSS, Journal of Computer and System Sciences, 18:35–46, 1981

[K.M. Van Hee, 1994] K.M. Van Hee: *Information Systems Engineering; a formal approach* Cambridge University Press, NY, 1994

[C.A. Petri, 1962] C.A. Petri: *Kommunikation mit Automaten* University of Bonn, 1962

[Baeten,Weijland, 1990] J. C. M. Baeten, W. P. Weijland: *Process Algebra*, Cambridge University Press, Cambridge, 1990.

[BPMN] Business Process Modeling Notation, BPMN 1.0, May 3 2004 Website: http://www.bpmn.org

[BPMI] Website: http://www.bpmi.org, accessed 30 April 2004

[PNML] M. Weber, E. Kindler: *Petri Net Markup Language,* Humboldt Universität, Berlin, 2002.

[EDOC] *UML Profile for Enterprise Distributed Object Computing:* Website: http://www.omg.org/technology/documents/formal/edoc.htm, accessed 24 May 2005.

[UML AD] R. Eshuis and R. Wieringa. *A formal semantics for UML activity diagrams – Formalising workflow models.* Technical Report CTIT-01-04, University of Twente, Department of Computer Science, 2001

[IDEF] Website: http://www.idef.com, accessed 10 January 2006

[EPC] August-Wilhelm Scheer, Markus Nüttgens: *ARIS Architecture and Reference Models for Business Process Management*, Institut für Wirtschaftsinformatik, Universität des Saarlandes, 2000

[Rummler-Brache] Website: http://www.rummler-brache.com/, accessed 10 January 2006

[UML] Unified Modeling Language, UML 1.5, Website: http://www.uml.org, accessed 1 May 2004

[BPEL4WS] Business Process Execution Language for Web Services, BPEL4WS 1.1, July 2002, Website: http://www.ibm.com/developerworks/library/ws-bpel/

[w3c] World Wide Web Consortium
Website: http://www.w3c.org, accessed 23 March 2005

[XML] Extensible Modeling Language, XML 1.1,
Website: http://www.w3.org/TR/2004/REC-xml11-20040204

[XSLT] Extensible Stylesheet Language Transformations, XSLT 1.0,
Website: http://www.w3.org/TR/1999/REC-xslt-19991116

[XPath] XPath 1.0
Website: http://www.w3.org/TR/1999/REC-xpath-19991116

[Petriweb] Website: http://www.petriweb.org, accessed 10 August 2004