PESOA

Process Family Engineering in Service-Oriented Applications

Process Modeling Techniques

Authors:
Arnd Schnieders
Frank Puhlmann
Mathias Weske

Table of Contents

Abstract

This paper introduces three process modeling languages for representing a system's dynamics, which are UML activity diagrams, Petri nets, and the emerging Business Process Modeling Notation (BPMN). Petri nets already exist for several decades and still enjoy great popularity among others due to their precise mathematical semantics and their straightforwardness. Activity diagrams are part of the UML specification and have changed considerably with UML version 2.0 having become more expressive and providing significantly more modeling elements. BPMN is a quite new process modeling language that aims to be sufficiently abstract and free of technical details in order to be usable also by business people for modeling their business processes.

1 Introduction

In today's dynamic business environments, change is the rule rather than the exception, both with respect to application level business requirements and technical requirements, regarding software systems that implement many functions required by modern organizations. While in general there might be many reasons for change, new business requirements and modified organizational and technological environments of a company and its partners are among the main sources of change requirements. Computer science in general and software engineering in particular have developed numerous approaches and techniques to deal with these issues and to enhance the flexibility of software systems, ranging from data independence in database systems to software design and analysis, software component technology, and object-oriented middleware and associated approaches.

Many of the products and services a company provides are produced by business processes, for instance processing an insurance claim or processing a credit application. For these types of business processes, workflow management systems have been developed. Using these systems, business processes can be modeled, and their execution can be controlled. Typically, a step in a workflow process is performed by an individual or by invoking an application. By controlling process executions, business processes can be performed faster, more reliably and more economically. However, process technology can also be used to model processes that are executed within software systems. This is already done in Enterprise Resource Planning systems where many business processes are pre-defined in the system. To represent the particular needs of organizations, these processes can be customized in many ways. These applications of workflow technology in software systems are called embedded workflow. Just like traditional workflow allows to model business processes that are being executed in a controlled fashion, embedded workflow allows to flexibly configuring software systems. To generalize these considerations, explicit knowledge on the processes that are executed in software systems is an important goal for the design and development of complex software systems, since it allows dealing with changes in a flexible way. Once the processes have been specified explicitly, they can be modified with rather limited effort, since process structures can be changed graphically and no software code needs to be re-written to implement the required changes.

In the context of object-oriented design and analysis, process modeling is often characterized by behavior modeling, and many concepts, methods and notations have been developed to model the behavior of complex software

systems. In this paper we provide an overview of recent techniques in behavior modeling and process modeling. The remainder of this paper is organized as follows: Section 2 introduces the current version of Activity Diagrams that have been developed in the context of the Unified Modeling Language. Section 3 discusses Petri nets that have been around since about forty years and that have been successfully used to model the behavior of complex systems in many different domains. In Section 4 we take a look at a more recent development, the Business Process Modeling Notation. This notation is currently being developed in the context of Web Services and related software technologies, where process technology is used to create business processes by composing existing Web Services. Concluding remarks and a brief evaluation and comparison of the process modeling techniques complete this paper.

This technical report was prepared during the first phase of the PESOA project (Process Family Engineering in Service-Oriented Applications), which is supported by the German Ministry of Education and Research. The report provides the basis for process modeling in PESOA.

2   UML Activity Diagrams

In software development decomposing complex problems in a number of simpler problems – also known as divide and conquer - is an essential principle to overcome the complexity of large systems. In object oriented modeling at least two different types of decomposition of a given software system can be done, regarding its structural and behavioral aspects, respectively.

The structural view on a software system points out "what" shall be provided by the system (e.g. which tasks, activities, etc.) reflecting a system's unchangeable aspects. Based on the fundamentals of the object-oriented approach like classification, the structural view provides for example information about inheritance, association, and aggregation relationships between the participating classes.

The behavioral view on the other hand shows "how" the system acts in order to fulfill a task. It thus provides information about the dynamic behavior of the system. There exist various languages for modeling system behavior. Some of the most popular ones comprise Petri nets, dataflow diagrams, several types of diagrams developed in the context of the Unified Modeling Language (UML), as well as formal approaches like, for instance, process algebras.

From which perspective a system is seen, whether from the static or from the dynamic point of view, depends on the particular intention of the modeler. In this paper we will focus on the description of the behavior of object oriented systems.

This chapter will deal with UML 2.0 activity diagrams as specified in [10]. In comparison to previous versions of the Unified Modeling Language, UML 2.0 activity diagrams have been changed in some major aspects with the aim of making activity diagrams more expressive and more intuitively manageable. Previously activity diagrams were a special kind of state diagram, with the difference that in activity diagrams an operation state changes upon completion of activity executions, while in state diagrams the state change happens due to the occurrence of an external event. Nevertheless UML 2.0 activity diagrams are still capable of modeling reactions to external events.

In UML 2.0 the activity diagram semantics is oriented at the Petri net semantics with activities and actions that produce and consume tokens rather than on state charts. However, expressed in simple terms, it is not possible to entirely map UML activity diagrams to Petri nets since, while Petri nets model

closed and active systems, which don't interact with their environment, activity diagrams may react to events occurring in their environment, as pointed out in [8]. In UML 2.0 the number of flows that can be modeled by activity diagrams is widened. It is now possible for example to model concurrent flows. Also the explicit modeling of control and object flows is new in UML 2.0 and replaces the use of state transitions in previous versions of UML activity diagrams [9].

Some of these enhancements of activity diagrams in UML 2.0 have been reached by integrating the UML activity model with the UML action model. The UML 2.0 activity and action model will be introduced in the following two chapters. Section 2.4 will then give a real world example for an e-business process described by means of a UML activity diagram. Chapter 2.4 is followed by a conclusion chapter summarizing the main properties of activity diagrams as well as their advantages and disadvantages.

2.1   UML Activities

In this section, the main elements of UML Activity Diagrams are discussed, including core elements, the handling of token and various routing constructs.

2.1.1   Core Elements

Generally speaking activity diagrams consider two dimensions of a process: the flow of data within the process and the flow of control coordinating the execution sequence of the tasks involved. These two aspects are also referred to as the control flow model and data flow model, respectively.

Figure 1:   **Structure diagram for core elements**
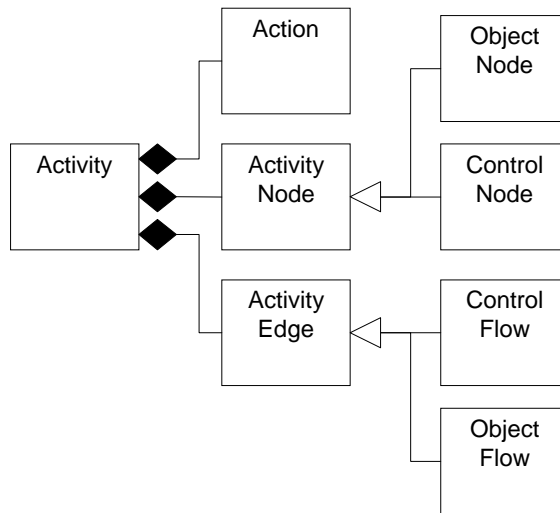
In order to clarify this, consider the following example. Let us assume that two tasks have to be carried out within the processing of an order, which are the delivery of the ordered products and the invoicing. Now the invoicing shall not be induced before the delivery of the products has been completed, which has to be made sure by the flow of control within the process. Tasks are depicted as a round-cornered rectangle and the flow of control between two tasks is shown as a directed arc. A diagram representing the processing of an order looks as follows.

Figure 2:   **Flow of control**

The diagram above still lacks from some important information, which is the flow of data between the two tasks. In fact the invoicing cannot be performed without some information about the products purchased by the customer. Since this information must be available to the task responsible for the delivery of the products the simplest way of providing this information to the task responsible for invoicing would be to pass it from one task to the other. Assuming that data handed from one task to another is indicated as a small square above the control flow arc the diagram displayed above expanded by object flow information would look as displayed in Figure 3.

Figure 3:   **Object flow**

```
┌─────────────┐      □      ┌─────────────┐
│   Deliver   │──────────▶  │    Send     │
│  Products   │             │   Invoice   │
└─────────────┘             └─────────────┘
```

An activity diagram and its control and data flow are represented by an activity, which typically contains activity nodes and actions interconnected by activity edges. Activity nodes comprise constructs for controlling the data and control flow, as well as nodes for indicating the presence of objects and data at certain points in the diagram. While the former nodes are called control nodes the latter are referred to as object nodes.
Additionally activity diagrams typically contain nodes for performing concrete calculations on the process' data. These processing-nodes comprise actions and behaviors whereas actions represent predefined calculations provided by UML while behaviors are user-defined.

Activities may be nested, thus being recursively resolvable into sub-activities, which on their part may consist of sub-activities again, and so on. Ultimately activities can be resolved into individual actions somehow interlinked with each other by means of edges and activity nodes according to the data and control flow. Actions are atomic insofar as they may not be decomposed further, but which may nevertheless be responsible for the invocation of other (structured) activities.

An activity may be assigned to a context element, called classifier. In this case the activity describes the behavior of that element [5]. Activity nodes and activity edges now are so-called RedefinableElements that can be specialized and replaced by more specialized elements if they are defined in the context of a classifier [10]. Specialization means the mechanism by which a more specialized classifier incorporates structure and behavior of a more general classifier [10].

Activity edges can be subdivided into control flow edges and object flow edges. Control flow edges start an activity node after the previous one is finished by means of the control tokens they carry thus realizing the control flow within the activity diagram. Object flow edges transport data in the form of object tokens thereby realizing the data flow within the activity diagram.
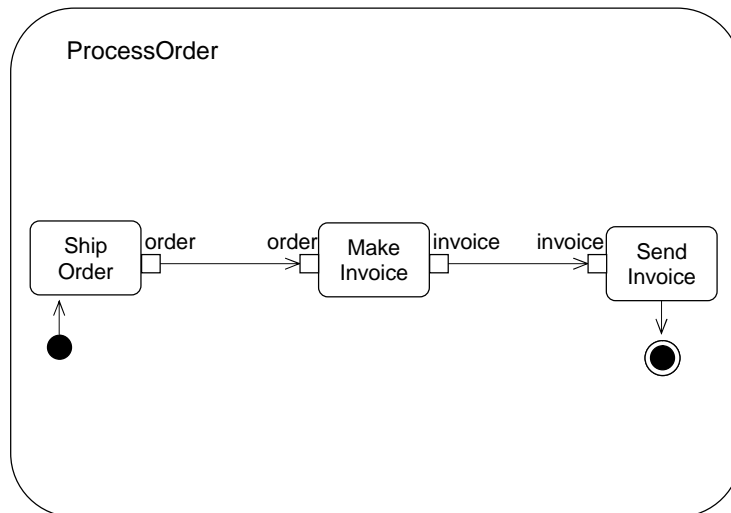
The name of the exemplary activity illustrated above is "ProcessOrder". It contains three actions, which deliver the ordered product items to the customer (represented by the "ShipOrder" action) and perform the invoicing ("MakeInvoice" and "SendInvoice" actions). Both the "ShipOrder" and "MakeInvoice" action and the "MakeInvoice" and "SendInvoice" partitions are connected through an object flow edge. A control flow edge looks exactly like an object flow edge with the only difference that the two small squares at the ends of the edge and the small identifiers close to the squares are missing. The small squares at the ends of the edges are called pins and will be introduced later in detail. For now it is only necessary to know that pins represent placeholders for input and output objects of actions, and that they may be of different types. Pins may also be omitted in the notation even though they are actually present in the diagram.

The two black dots close to the "Ship Order" and "Send Invoice" action are an initial and final node, which indicate the starting point and end point of the activity.

Altogether there are three alternative notations for an object flow edge (Figure 5). The first one corresponds to the notation used in the example above. In the second one the pins are elided and in the third one the object, which is passed from one action to another is represented as a large rectangle that divides the connecting edge. The latter notation is only allowed if the pins at the source as well as at the target node are of the same type.

Figure 5: **Notations for object flow edges**
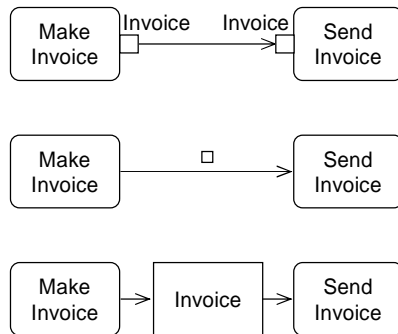
2.1.2 Execution of Activities

Tokens control the execution flow of nodes within an activity diagram. Tokens may contain objects. Additionally they may just be utilized for passing on the control from one action to another. Tokens carrying objects also carry control information. An action accepts tokens from its input edges and starts execution, except where noted, as soon as all of the incoming object flow edges provide the required input objects and all incoming control edges have tokens (implicit join). The tokens are then accepted from all of the input edges and placed on the node. After completion of the execution the control and object tokens are taken from the node and offered to all of the outgoing control edges and pins (implicit fork).

In order for a token to be allowed to traverse an edge it has to fulfill the token flow rules not only for the target node, but also for the source node and the connecting edge. So even if a source node offers a token, the token may still be prevented from moving to the target node because of the rejection by the edge or the target node.

Object flow edges offer various mechanisms for modeling token flow rules. Tokens that want to traverse an object flow edge can be evaluated by guards, which describe traversal constraints for tokens and which have to be evaluated to true for a token to be allowed to traverse the edge. Additionally weight attributes define the minimal number of tokens, which have to traverse an edge concurrently. As soon as there are sufficient tokens in terms of the edge weight offered by the source node to the edge all of the tokens at the source node are offered to the target node at once. In the broader sense selection behaviors for object flow edges can be considered as a modeling instrument for token flow rules as well, since a selection behavior may determine in what sequence incoming tokens are allowed to traverse the edge. In contrast to object flow edges control flow edges forward whatever control token being offered by the source node, to the target node so they don't possess any means for accomplishing token flow rules.

Nodes provide some instruments for modeling token flow rules for source nodes as well as for target nodes. The kinds of restriction mechanisms they offer depend on the type of node. For actions pre-conditions and post-conditions may be defined that must hold for an action to begin and to terminate processing. Object nodes may accept only objects of a certain type and with a certain state. Moreover they may have a limit resulting in the object node to reject an object token offered by the incoming edge in case the limit of the object node is reached. Additionally object nodes can be ordered according to one of the predefined orders (called object node ordering kinds) or by means of a selection behavior that defines the order in which tokens are offered to the outgoing edge. An example for token flow rules for object nodes is given later in this text. Concerning control nodes only join nodes may define token flow rules by means of join specifications. These can be used to define under which conditions tokens are accepted by the incoming edges and offered to the outgoing edge. Join nodes will be discussed in detail later.

We now take a look at the example depicted in Figure 6 that illustrates the abovementioned modeling instruments weights and guards. Let us assume that we want to build a new football team. Since a football team always consists of at least eleven football players a respective weight attribute at the incoming object flow edge of the "FormFootballTeam" action makes sure that the football team isn't built as long as there are not at least eleven players available. The "FormFootballTeam" action and "PlayFootballMatch" action are interlinked by another object flow edge. This time the edge disposes of a guard, which makes sure that the "PlayFootballMatch" action isn't activated as long as the date of the match hasn't been reached.

Figure 6:   **Weight and guard adornments**
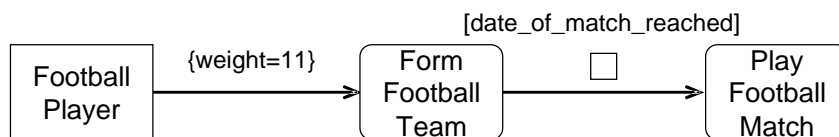
2.1.3 Object Flow

Object flow edges are responsible for transferring data to or from object nodes. They may have an executable node at most at one end. Two object nodes may also be connected by object flow edges with an intervening control node. A control node is an element for steering the routing of the data and control flow and will be described more in detail later.

**Figure 7:**   **Structure diagram for object flow elements**
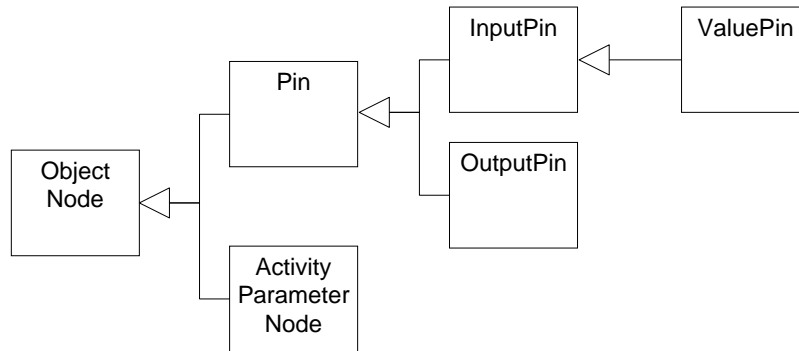
The flow of data within an activity diagram can be supported by object nodes, which indicate the availability of data at a particular point in the diagram. They are typically located at the end of an edge and act as an input/output buffer for an executable node, collecting tokens as they are waiting to be consumed by the node or to traverse the next edge. The data values of the tokens thereby have to conform to the type of the object node (if a type is specified, else the contained values may be of any type) and have to be in the required state (if demanded). An object node may hold tokens up to the specified limit (in case the object node possesses such a limit).

As mentioned above object nodes are used in particular for buffering the input and output data of executable nodes. Object nodes, which serve as the input and output buffer of an action are called "pins", whereas object nodes assigned to activities are called "activity parameter nodes". The objects held by the pins serve as the inputs and outputs of an action being matched to its input and output parameters.

Figure 8 shows an example for the pin notation. The two small squares at the "TicketSale" action are pins for data concerning the departure and arrival of the flight. The ticket sale won't start until the "Arrival" and "Departure" pin don't provide this information to the "TicketSale" action.

Figure 8:   **Pin notation**
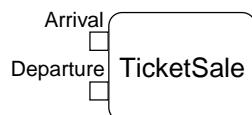
In contrast to pins, activity parameter nodes always only dispose of either an outgoing edge and no incoming edge (input activity parameter node) or an incoming edge but no outgoing edge (output activity parameter node). The parameters provided to the activity by an activity parameter node can be accessed by any action within the activity. An activity only provides tokens on its outgoing activity parameter nodes as soon as there are no tokens left circulating inside the activity.

Mutually exclusive alternative sets of inputs and outputs a behavior may use can be modeled using the core elements. Normally several edges entering a behavior stand for an "and"-condition, which means, that the input data on all edges has to be available for the behavior to start. Likewise multiple outgoing edges of an activity normally mean that tokens will be placed on any of the outgoing edges after an activity has terminated. Now using core elements it is possible to model alternative groups of input and output parameters. For input, when one group or another has a complete set of input flows, the activity may begin. For output, the group of output flows that may occur depends on the internal processing of the behavior.

Output parameters of activities may have an additional isException property controlled by means of a corresponding attribute. Attributes in general are values, which are available to a node during execution. They are therefore suitable for adjusting certain properties of a node. The type of attributes a node possesses depends on the kind of node. If the isException attribute is set for the output parameter of an activity, this means that all flows within the activity shall be aborted as soon as a token is issued by the output parameter with the isException attribute set. If the isException attribute is set for an output parameter of an action the issue of a token by this parameter prevents any other output parameter of the action to emit any further token.

Figure 9 is an adapted example taken from [5] showing the notation of activity parameter nodes, core elements and the isException attribute notation for activity parameter nodes and actions. The mentioned elements are illustrated on the basis of the payment process for a plane ticket starting with the reception of an invoice (provided by the "Invoice" incoming activity parameter node) and resulting in the issuing of the corresponding flying ticket (via the outgoing activity parameter node "Ticket").

The customer may accomplish the actual payment using his or her credit card or by means of a bank transfer. Accordingly the processing of the payment ("Process Payment") requires either the complete credit card information (represented by the objects "Number", "Name" and "Date") or a money order ("MoneyOrder"). The fact that alternatively the complete credit card information or the money order is required in order to start the process payment is expressed using core elements. Considering for example the outgoing core elements at the "Make Payment" action one core element com-

prises the credit card information, while the other one holds the money order.

The "ProcessPayment" action disposes of an outgoing pin with the isException attribute set indicated by the small triangle. Thus the "ProcessPayment" action issues an error whenever an exception occurs during the processing (e.g. the specified credit card is invalid). In this case the "Payment" activity will issue an error value instead of a ticket as indicated by the "Error" activity parameter node with the isException attribute set (again a small triangle written close to the activity parameter node).

Figure 9: **Activity parameter nodes, isException attribute, and core element**
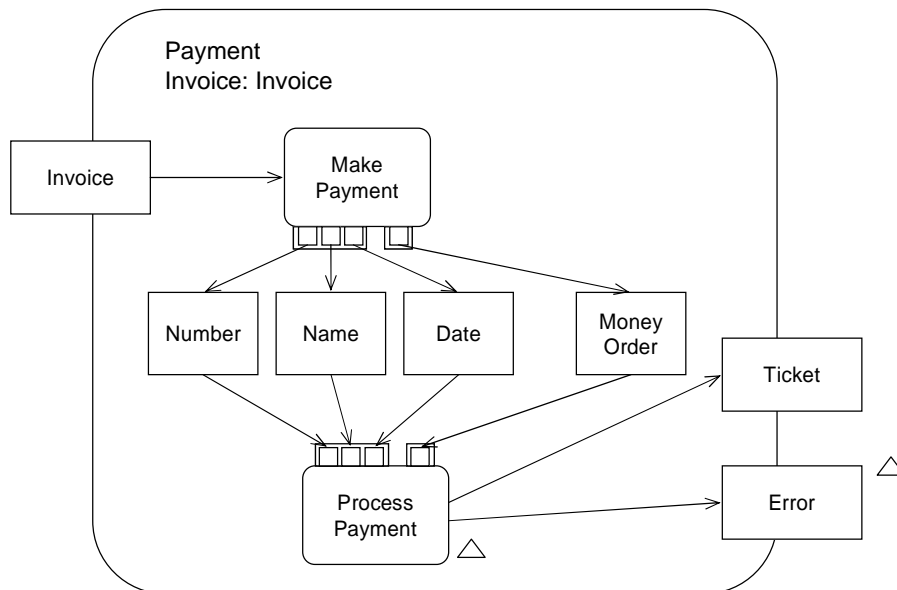
Object flow edges can exhibit a so-called token selection behavior. A token selection behavior can select tokens to be offered to the target node from a number of incoming tokens according to specified rules.

The diagram in Figure 10 shows an adapted example for an object flow selection behavior taken from [10]. "Fill Order" offers object tokens of type "Order" in the state "Filled" to the outgoing edge via its outgoing pin. The objects waiting for traversal are investigated by the edge and the objects are selected in the order of their priority. If there are two orders with the same priority the one which arrived first is chosen.

16
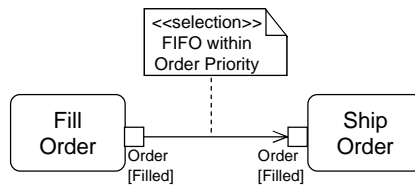
2.1.4 Storage of Tokens

A data store node is a special kind of central buffer node, which is used to model a central buffer for persistent information. It keeps all of the entering tokens and makes a copy of them whenever they are supposed to move downstream. Objects residing in the data store node are overwritten by identical new ones, which enter the node.

Figure 11: **Structure diagram for storage elements**
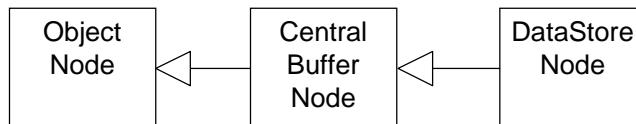
As visible in Figure 12 data store nodes are indicated by the keyword "datastore". In this example derived from an example in [10] hired employees are stored persistently in a data store node. They remain there but can be retrieved from it by means of a selection behavior. In our case employees without any duties so far are read from the personnel database as soon as they shall be assigned with a new task.
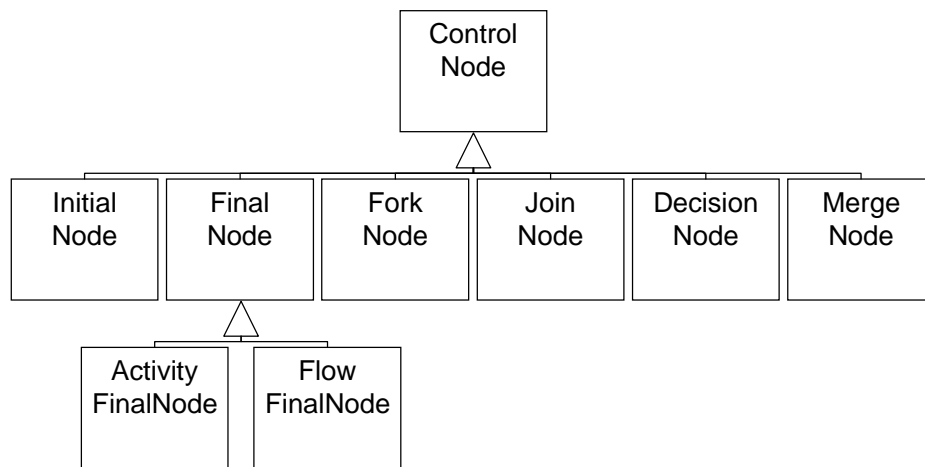
Figure 12: **Data store node**
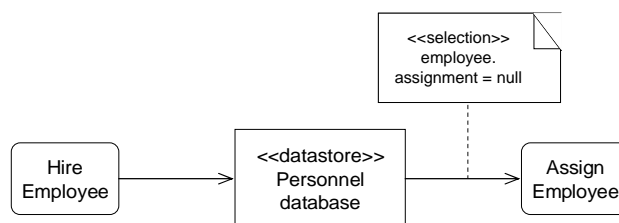
2.1.5 Routing the Control and Object Flow

The flow of control and data within an activity diagram is modeled with the aid of control nodes, which serve as kinds of switches in so far as tokens are not allowed to rest at control nodes. The consequence is that tokens, which are rejected by the outgoing edge, are discarded instead of being saved for another attempt. Control nodes comprise fork nodes, join nodes, decision nodes, merge nodes, initial nodes, and final nodes.

Figure 13: **Structure diagram for elements for steering the control and data flow**

```
                          ┌──────────┐
                          │ Control  │
                          │  Node    │
                          └────△─────┘
   ┌───────┬───────┬────────┼────────┬─────────┬────────┐
┌──────┐┌──────┐┌──────┐┌──────┐┌─────────┐┌──────┐
│Initial││ Final ││ Fork  ││ Join  ││Decision ││ Merge │
│ Node  ││ Node  ││ Node  ││ Node  ││  Node   ││ Node  │
└───────┘└──△───┘└───────┘└───────┘└─────────┘└───────┘
       ┌─────┴─────┐
  ┌─────────┐┌──────────┐
  │ Activity ││   Flow   │
  │FinalNode ││FinalNode │
  └──────────┘└──────────┘
```

Parallelism in activities can be modeled using fork and join nodes. Here parallel execution of activities doesn't mean that they must be carried out synchronously but only that they are independent from each other and hence may also be processes at the same time.

Fork nodes have one incoming edge and multiple outgoing edges. Incoming tokens are copied and passed to every outgoing edge whose token flow rules currently allow the acceptance of a control token.

Join nodes are the counterparts of fork nodes, i.e. they have several incoming edges but only one outgoing edge. The so-called join specifications define according to which rules a join node offers tokens to the outgoing edge. Therefore it is proved for every incoming edge whether it offers a token. It is then evaluated on the basis of the token configuration of the incoming edges, if it conforms to a Boolean value specification representing the join specification of the join node and that refers to the names of the incoming edges. The default is that tokens are offered to the outgoing edges as soon as at least one token is being offered at each incoming edge. Apart from the already mentioned join specifications possible deviations from this rule may

also result from the characteristics of the outgoing edge and its target. The outgoing edge for example may meet an object node that has reached its upper bound and thus rejects any incoming tokens.

As soon as the join specification of the join node is fulfilled tokens are offered on the outgoing edge according to the following join rules: if the incoming tokens are all control token, then one token is offered to the outgoing edge. If some of the incoming tokens are control tokens while others are data tokens, then only the data tokens are offered to the outgoing edge.

Figure 14 presents the notation for join and fork nodes based on an example modeling a flight. After the passengers have checked in and boarded the flight starts. During the flight the passengers have a meal and watch a movie (parallelism of "watch movie" and "eat" actions enabled by the fork node – the vertical line with one incoming edge and two outgoing edges). Some time after the passengers have watched the movie and finished their meal the aircraft lands and the passengers are supposed to unboard.

Figure 14:   **Fork and join node**
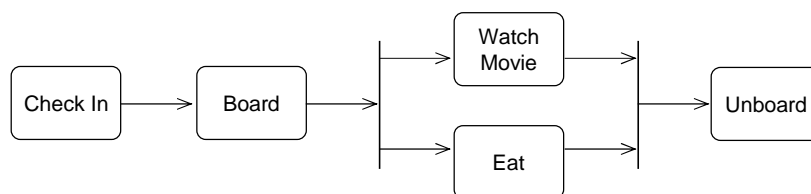
Alternative flows within an activity diagram are modeled by means of decision and merge nodes. Decision nodes are control nodes with only one incoming edge and multiple outgoing edges. They are used to determine to which one of a number of alternative outgoing edges a token shall be offered.

The process of selecting the outgoing edge is performed via guards. A special else-guard can be assigned to an edge to which a token is offered in case no other guard has been evaluated to true. There shall always be only one traversable edge in the end, since decision nodes do not duplicate tokens. Whether the unambiguousness is finally reached through the mutual exclusiveness of the guard definitions or through a combination of guard decisions and input conditions of the outgoing edges, doesn't matter. Before the guards are evaluated a token can be passed to the decision input behavior of the node (if specified), which can avoid redundant recalculations of conditions in guards. Figure 15 will give an example of how such a decision behavior is modeled.

Merge nodes are the counterparts of decision nodes, as they bring together multiple flows. There is no joining of tokens and every token, which enters the merge node through any of the incoming edges, is forwarded to the outgoing edge. Initial and final nodes indicate the start and end of a flow, respectively. Activities are invoked at their initial nodes. Therefore to start an activity a token is placed at its initial nodes. In contrast to other control nodes initial nodes save tokens that are not consumed immediately by the outgoing edge.

There are two types of nodes modeling the end of flows in activity diagrams both derived from the final node element: the activity final node and the flow final node. The characteristics of an activity final node is that it terminates all flows within the activity as soon as it is reached by a token thus providing a means to model non-local termination of all flows in an activity. Flow final nodes in contrast terminate flows by destroying arriving tokens while the other flows within the same activity remain unaffected.

In Figure 15 the notation for initial node, decision node, merge node, and activity final node is shown. That diagram shows parts of the dispatching of a flight. The diagram starts at the "Buy Ticket" action as indicated by the starting node (the black dot). If a passenger disposes of the plane ticket he may proceed to the check in. At the decision node (the diamond symbol) an input behavior first detects whether the flying ticket is a first class or economy ticket. If it's an economy ticket the passenger proceeds to the normal check in and to a waiting room afterwards. If it's a first class ticket the passenger is dispatched at the first class check in and may wait in the lounge afterwards. The following step is identical for first class and for economy class passengers. Thus the token flow is merged by a merge node (the diamond shape with the two incoming edges and one outgoing edge). The outgoing edge of merge node disposes of a guard, which makes sure that the boarding starts only as soon as the gate is opened. The process ends after the completion of the boarding process, which is indicated by an activity final node (the black dot inside a circle).

Figure 15:   **Initial node, decision node, merge node, and activity final node**
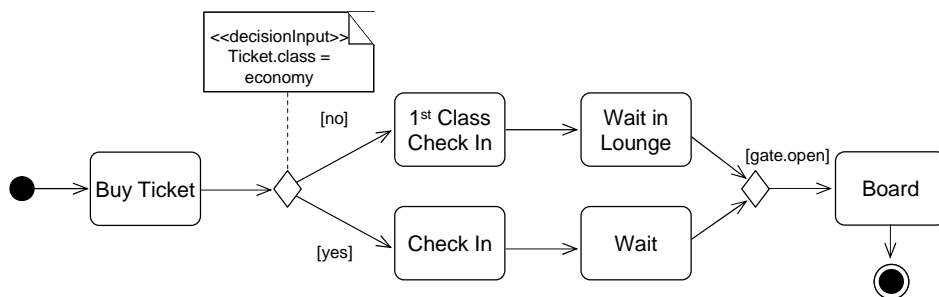
Figure 16 shows an example taken from [10] that illustrates the utilization of a flow final node (the cross in a circle at the right side of the figure). The particularity here is that due to the utilization of a flow final node it may happen that after a token has reached the final node and there are no more components being built the "Install Component" action can still be active.
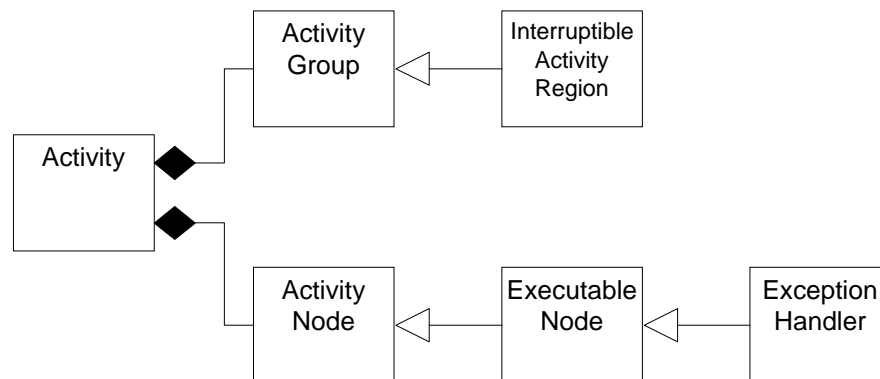
Figure 16:   **Flow final node**
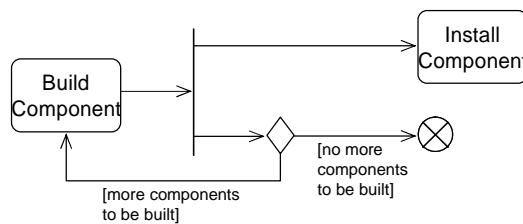
2.1.6 Deviations from the Normal Flow of Control

Two means of disrupting the normal flow of control are interruptible activity regions and exceptions. In this survey only exception handlers will be regarded.

Figure 17:   **Structure diagram for elements for modeling deviations from the normal flow of control**

Exceptions within an activity diagram are processed by exception handlers. An exception handler is an element, which is assigned to a node protected by the handler and which disposes of a body that is executed in case an exception of the handled type occurs at the protected node. If there is no handler defined, which matches the exception, it propagates to the enclosing activity until it is finally caught or until it reaches the topmost level of the system. If an exception propagates out of the protected, nested node, all tokens within that node are terminated. Since an exception handler doesn't dispose

21

of any incoming or outgoing edges by itself it inherits the incoming and out-going edges of its protected node.

The result tokens of the exception handler body become the result tokens of the protected node. When the execution body completes execution, it is as if the protected node had completed execution.

Figure 18 shows how an exception can be handled. A payment is normally supposed to be performed by credit card. Therefore a customer hands over his or her credit card within the "Pay With Credit Card" action. Thereupon the credit card information is passed to the "Process Payment" action, where the money is debited from the customer's account. If a customer doesn't dispose of a credit card a "NoCard" exception is raised, passed to the re-spective exception handler (the rectangle containing the "Bank Transfer" ac-tion) and a bank transfer is initiated. After the bank transfer has terminated the payment continues with the execution of the "Process Payment" action.

Figure 18:   **Exception handling**
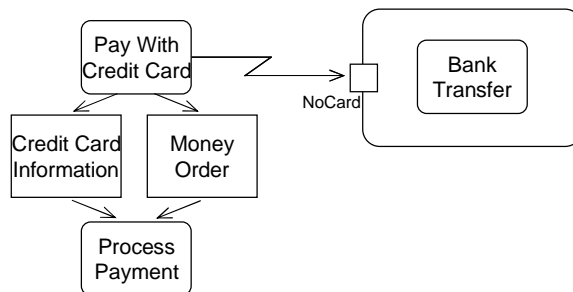
2.1.7   Additional Instruments for Controlling Flows

The instruments for advanced steering of the control and data flow intro-duced here are all features of specializations of the structured activity node, which is a structured portion of an activity that doesn't issue any output to-kens until all embedded nodes have completed execution (i.e. no tokens are left inside the structured activity node). On the other hand no node within the structured activity node starts execution until all input pins and incoming control flow edges of the structured activity node have tokens. Therefore in order to accept inputs and provide outputs structured activity nodes can ac-cess the pins and control flows attached to it. Any activity edge belonging to the structured activity node must have its source and target node inside the structured activity node, i.e. no object or control flow edge may lead from outside the structured activity node inside or vice versa.
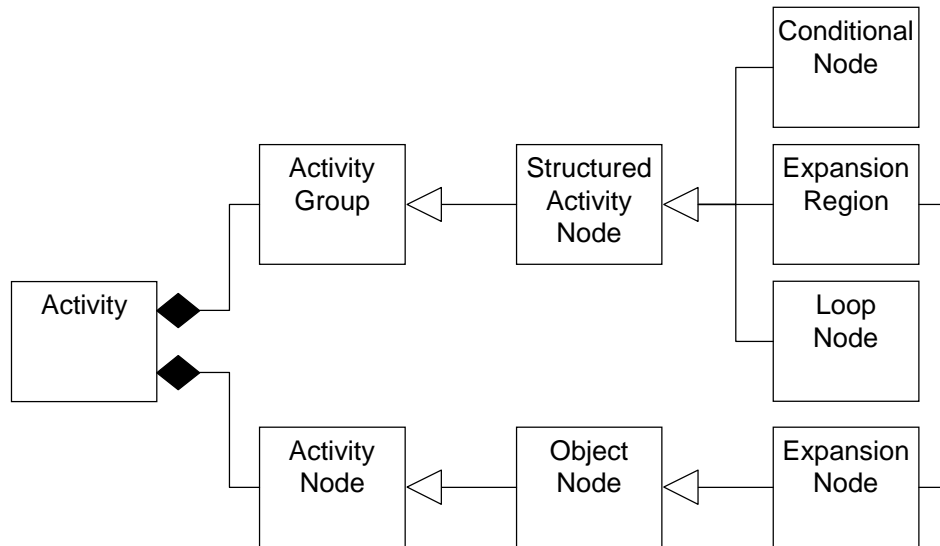
Structured activity nodes may have variables, which allow different activities within the structured activity node to access the data held by the variable through read and write operations.

A loop node is an activity node that can be executed multiple times. Even though modeling loops using the standard activity diagram elements is quite straightforward as indicated in Figure 20, with loop nodes activity diagrams provide a means for representing iterative execution of actions and behaviors in a more structured way. Moreover loop nodes inherit the properties of structured activity nodes, which may be desired as well.

Figure 20: **"Do while" and "while" loop**

23
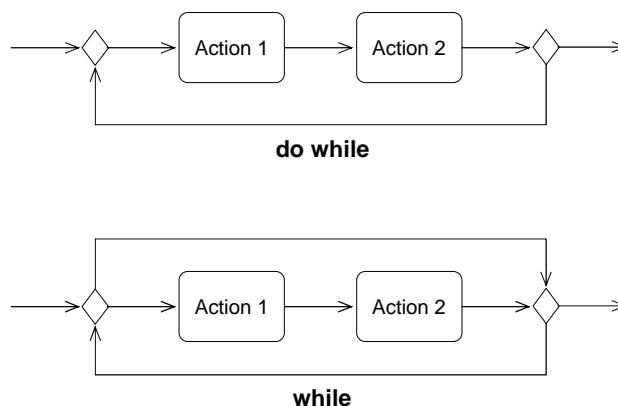
A loop node consists of three subregions, the setup section, a body section and a test section. The body section is executed as long as the test section returns a true. The results of the final execution of the body are made available via pins after completion of the loop.

The setup section is executed once on entry of the loop. A control token is thereby offered by the loop node to every front end node (= a node without any predecessors) within the setup section. Additionally the elements within the setup section of the loop may have individual (typically data flow) dependencies to nodes outside the loop. The setup section is finished when all of the backend nodes (nodes without any successor within that section) have completed execution.

When the setup section has terminated or the test section (which may come before or after the body section) has completed execution and produced a true value, the body section is executed which terminates as soon as all backend nodes have terminated. The results of the last execution are then provided to the successor nodes of the loop via output pins.

The test section disposes of a certain decider pin that indicates after the fulfillment of the test section whether the body section shall be executed again.

Figure 21 shows an example for a loop-node. The setup-section of the loop node is indicated by the keyword "for", the test-section by the "while"-keyword, and the body-section by the "do"-keyword in the upper left corners of the respective sections. In this example the overall sum of a list of incoming prices is calculated. For this before entering the loop-node the number of prices to sum up is determined. According to the number of prices the body of the loop node is then executed once for every subtotal.

Figure 21: **Example for loop node**
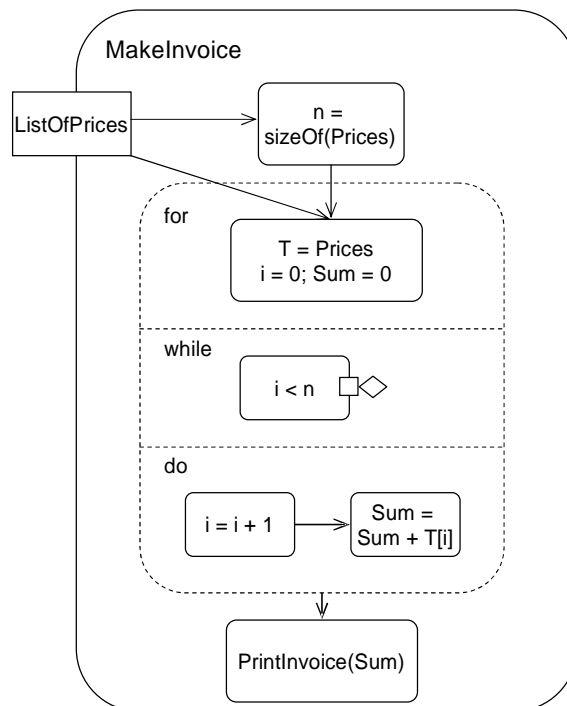
An expansion region is a structured activity region that executes multiple times according to the number of elements in an input collection. The expansion region is executed once per element in the input collection as long as there remain elements.

The input/output collections of an expansion region are modeled as expansion nodes, which are special kinds of object nodes. These are broken into their individual components inside the region, which is then executed once per element.

If an expansion region has more than one incoming pin all of the input collections at each pin must be of the same kind (expansion node type) and must hold the same number of elements. But the elements of the different input collections may have different types.

In case an expansion region has outputs, they must be collections of the same kind and must contain elements of the same type as the corresponding inputs.

There are three different ways of performing the multiple executions of the activity region:

- The executions may happen in parallel

- They may be performed iteratively with one execution being performed after the other. During every iteration one element from the collection is made available to the region. After each execution of the region one element is added to the output collection.

- The elements of the collection may be passed to the region as a stream with the effect that the region is executed only once. Therefore the region must be modeled to handle streams properly.

Figure 22 shows an expansion region calculating the prices for a collection of flight segments. This is necessary since a flight consists of at least two (the outward flight and the return flight) or more (if there are intermediate landings) flight segments. An expansion region is recognizable by the dashed line and by its input and output expansion nodes (the small rectangles divided by vertical bars into small compartments). The execution mode is written in the upper left corner of the expansion region.

Figure 22:   **Expansion region with multiple contained actions**
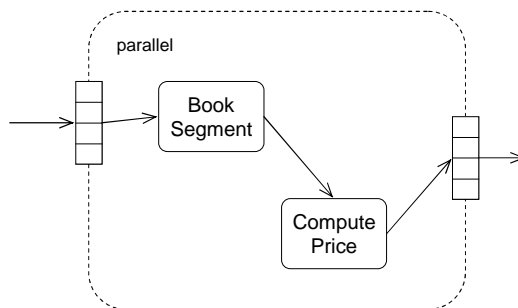
If an activity region contains only a single node, the shorthand notation depicted in Figure 23 can be applied. The activity region then has a continuous line and the name of the enclosed action is written in the middle of the expansion region. In this example a trip route is specified by means of a number of trip segments. These segments are issued as a collection by the "Specify Trip Route" action. For any of the sections the "Book Flight" expansion region books the respective flight.

Figure 23:   **Shorthand notation for expansion region containing single node**

2.1.8 Additional Means for Structuring Activities

With structured activity nodes one element for subdividing activities has already been introduced. Structured activity nodes are a specialized form of activity groups, which serve as a generic construct for grouping nodes and edges and which don't have any inherent semantic. Structured activity nodes are always notated with a dashed round cornered rectangle, which encloses its nodes and edges and which has the keyword "structured" at the top.

**Figure 24:** **Structure diagram for elements for structuring activities**
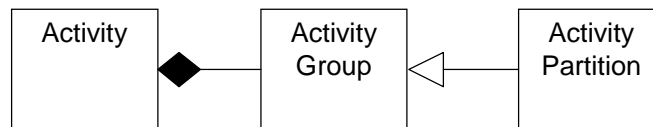
Another specialization of activity groups are activity partitions, which are used to group activities which have some communities. Activity partitions do not affect the token flow of the model, but they can constrain the activities contained in the partition and allow them to share contents.

Activity partitions can be multidimensional and swim lanes can express hierarchical partitioning. Single entities within a partition can be marked as being "external", i.e., as not being affected by the partition structure.

A structured partition with hierarchically nested swim lanes is shown in Figure 25. The sales department of an airline here is responsible for sending the invoices to the customers. A customer pays the invoice ("Make Payment" action) and sends his or her credit card data to the accounting department of the airline, which will process the payment ("Process Payment"). After the payment has been processed, the flying ticket is sent to the customer.

Figure 25:   **Hierarchical partition structure**
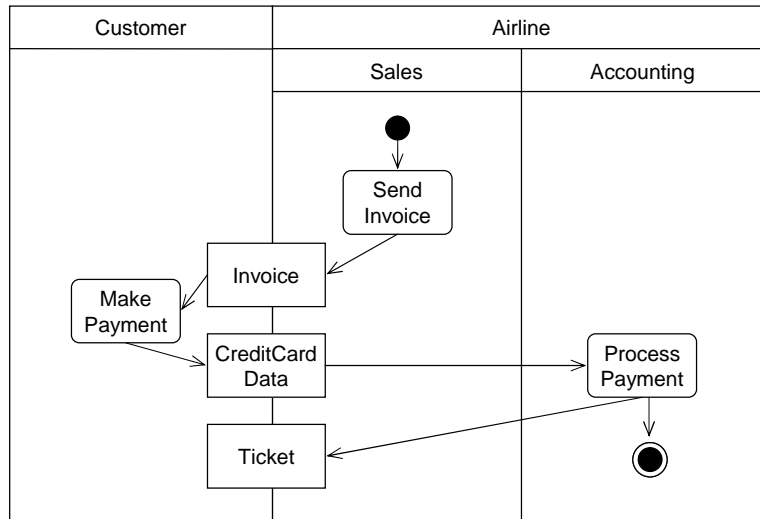
In Figure 26 you find an example for a multidimensional partition taken from [10]. With the "Make Payment" action it also shows an example for an action, which is physically contained within a swim cell without actually being part of it. This means in our case that even though the "Make Payment" class is contained within the Seattle/Accounting Clerk swim cell this process is not performed by an accounting clerk in Seattle but by a customer as indicated in brackets.

Figure 26:   **Multidimensional partitions and "external" nodes**
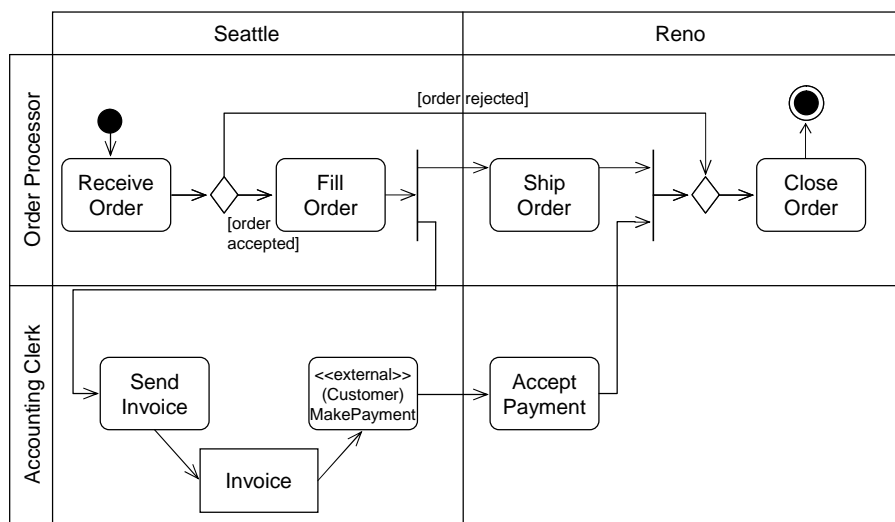
2.2 UML Actions

In activity diagrams as mentioned in the previous chapter actions represent atomic units of execution, which are not decomposable in further subunits. In contrast to user defined behaviors actions are processing units that are provided by UML.

Some actions can be used for invoking user-defined behaviors, others for getting the values of attributes or linking objects together.

"Call actions" are invocation actions that invoke behaviors. In case it's a synchronous invocation of a behavior the invocation action returns a return value.

In Figure 27 you see the notation for an activity being invoked by a call action indicated by the rake-style symbol within the action symbol, which shall suggest the hierarchical structure of an activity.

Figure 27:   **Invocation of an activity notation**

Activity name
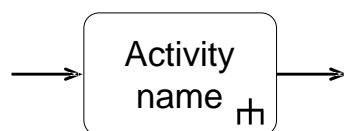
A "send signal action" asynchronously sends a signal instance, which is generated from the input data of the action, to a target object, where it may result in the execution of a contained behavior for example. Figure 28 shows the notation for the send signal action.

Figure 28:   **Send signal notation**

Signal Type
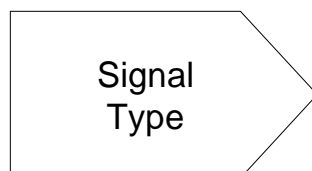
There are also actions, which are activated as soon as an event that matches certain conditions occurs. These actions are called "accept event

29

actions". Having received a matching event they issue a token describing the event. If the event is a signal the receiving accept event action is also called "accept signal action". If the event is a time event, it is also called a "wait time action". In the latter case the token issued by the action contains the time at which the event occurred. If the event is a change event or call event the result is a control token. An accept event action with no incoming edges is always enabled to accept event inputs. The figure below shows the notation for an accept signal action and for an accept time event action.

**Figure 29:**   **Accept signal action and accept time event action**
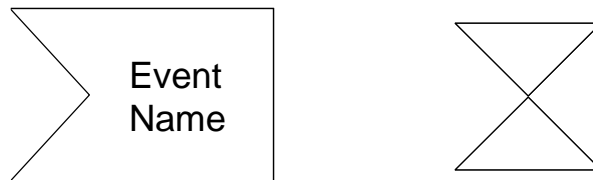
2.3   Example

This section will introduce an activity diagram for a real-world application. The system that shall demonstrate the application of activity diagrams is a distributed e-commerce platform for trading food.
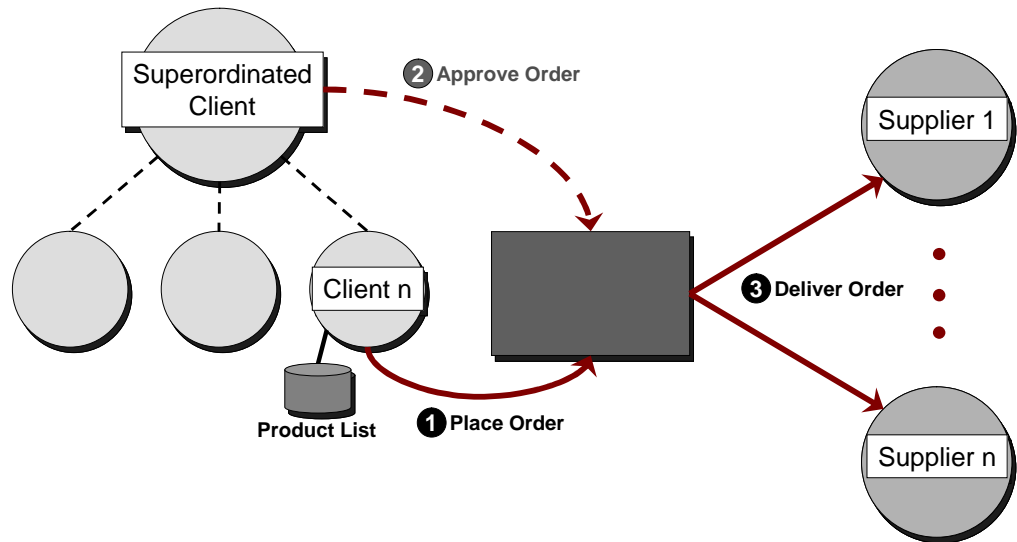
The system consists of program modules at a central server computer and modules that reside at the computers of the platform users. The platform modules at the users' computers are integrated with the users' ERP systems. The customer-sided modules communicate with the server-sided modules by means of SOAP transactions. Digital signatures and encryption protect the transactions.

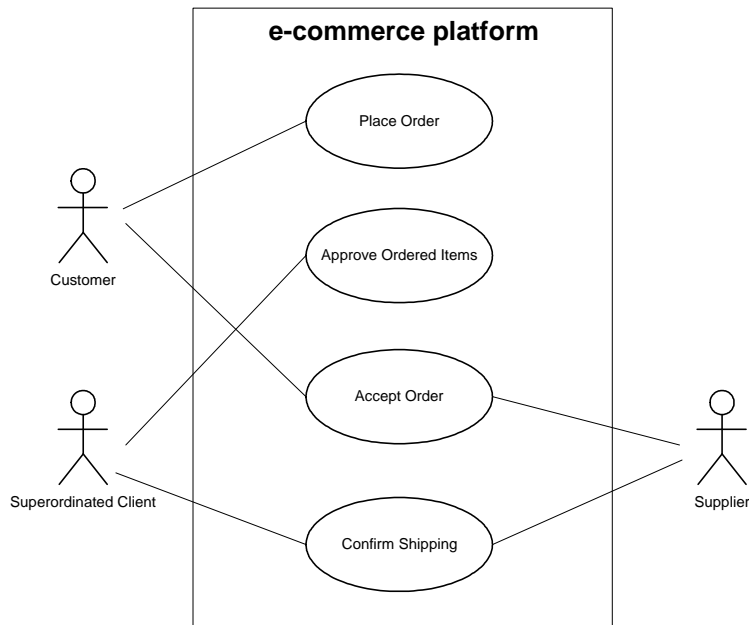**Figure 30:**   **E-commerce platform**
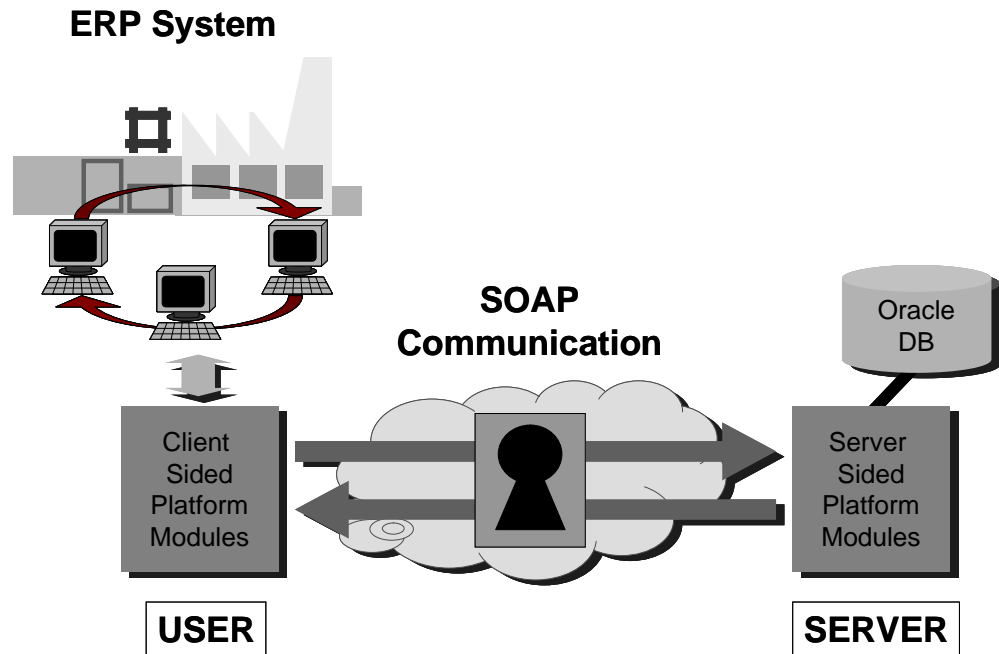
ERP System

Three different kinds of platform users can be distinguished, whose ERP systems are connected synchronously to the e-commerce platform. First of all there are food suppliers, which trade their products via the platform. The products offered by the suppliers are bought by customers, who locally maintain a list of available products. Clients may be supervised by clients, which act as shopping centers and which have to confirm any order placed by a client before the order is forwarded to the suppliers. An order may comprise products, which are provided by different suppliers. Thus the e-commerce platform is responsible for forwarding the different order sections to the correct suppliers.

Figure 31:   **Interaction between platform users**

Superordinated Client

2 Approve Order

Supplier 1

Client n

3 Deliver Order

Product List

1 Place Order

Supplier n

The e-commerce platform sketched here represents orders in XML, while any of the customers' ERP systems may use a different representation. Hence an order passed by the customer to the e-commerce platform first has to be converted from the customer specific format into the platform format in order to enable further processing.

e-commerce platform

- Place Order
- Approve Ordered Items
- Accept Order
- Confirm Shipping

Customer

Superordinated Client

Supplier

2.3.1 Place Order Use Case

Section: Main

Actor Action	System Response
1. This Use Case begins when the Customer's ERP system (CES) passes an order placement request to the system. The request is passed as a string in the customer-specific format and specifies the command to be processed (in this case an order placement) and the parameters required to carry out the requested command. In this case the parameters will be among others the items to purchase each with the ordered quantity, the name of the respective vendors and the customer id.	2. The System extracts the identification number of the requested service (which is an order placement) and the parameters (which are the ordered items, the corresponding suppliers and the customer id) belonging to it from the string passed by the CES making sure it involves a proper service request.

	3. It transforms the order data into the transport format adding a transaction identification number. Thereupon it encrypts and signs the data cryptographically. Furthermore a hash total is calculated to protect the transmitted data against undetected modifications.
	4. The order is sent to the platform server by invoking a respective SOAP service provided by the platform server and committing the protected order in the transport format as a parameter. If the server-sided platform module is not reachable, the customer-sided platform module proceeds according to "Retry Message Delivery".
	5. The platform server decrypts the transmitted data, checks the hash total and validates the appended signature. This process may involve the validation of the originator's public key. The order data is extracted from the transport format afterwards.
	6. It has to be proved now that the desired order is conformant to the access rights for this user (authorization check).
	7. The order is stored in an oracle database.
	8. Notify the CES that the requested service has been invoked successfully. This will be realized by means of a response message for the invoked SOAP service. If the customer-sided platform module is not reachable, the server-sided platform module proceeds according to "Retry

	Message Delivery".
	9. It may be the case for premises that the order has to be approved by its superordinated client. If so the respective approvals have to be awaited first before the processing can continue.
	10. As soon as the order approval has arrived the status of the order is changed accordingly. Now requests for the various products, which have been approved (or which haven't had to be approved at all), can be sent to the corresponding suppliers.
	11. Therefore the respective order is read from the database.
	12. The order has to be transformed into service requests (in transport format) holding the order fragments targeted at the different suppliers. Afterwards the usual security mechanisms have to be applied.
	13. The service requests containing the partial orders are sent to the re-spective suppliers as a SOAP re-quest. If one of the corresponding computer systems is not reachable the server-sided platform module proceeds according to section "Retry Message Delivery".
	14. At the supplier side the system again decrypts the request in trans-port format and validates the hash total and the signature. This process may involve the validation of the originator's public key.
	15. Afterwards the identification number of the platform server initi-

Actor Action	System Response
	ated service request is extracted from the request data and the service parameters are passed to the module implementing the delivery of the order.
	16. This module extracts the order data from the transformation format, packs it into the customer-specific format and passes it to the CES*.
	17. Notify the platform-server that the (partial) order has been delivered successfully. If the server-sided platform module is not reachable, the supplier-sided platform module proceeds according to "Retry Message Delivery".

Section: Retry Message Delivery

Actor Action	System Response
	1. The sender-sided platform module retries to dispatch the SOAP message again after five minutes.
	2. If the delivery of the SOAP message to the receiver-sided platform module fails again, the server-sided module proceeds with step 1., unless the delivery has failed already five times. In this case the sender-sided module abandons to dispatch the message.

The figures below show an activity diagram representing the place order use case described above.

The customer-sided processing of the place order transaction is depicted in the first diagram. If the "Send SOAP Request" activity doesn't succeed in
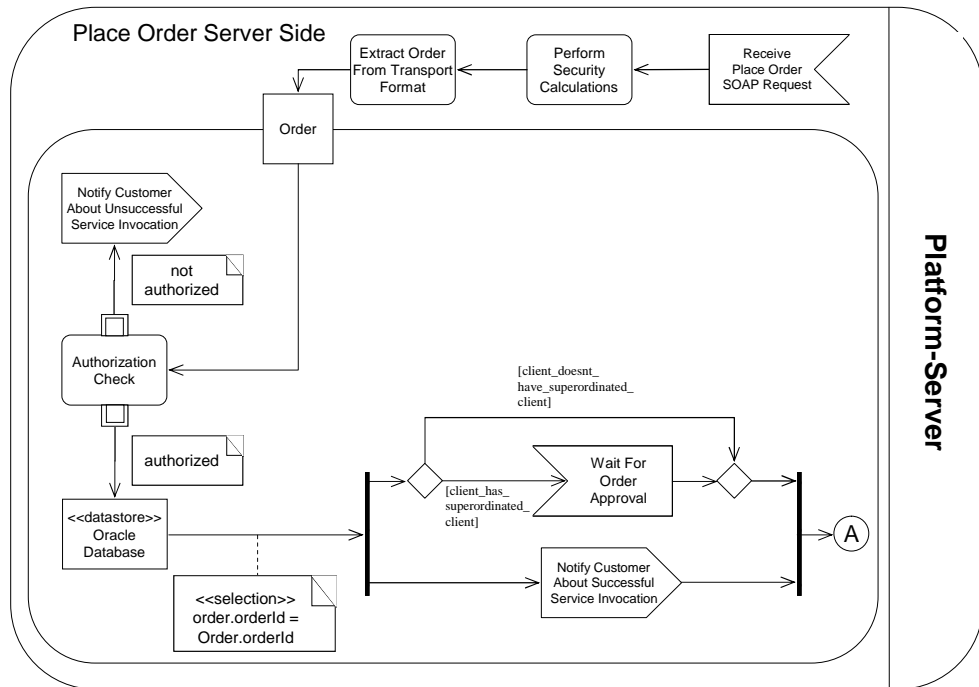
sending the order to the platform-server, the delivery of the order is abandoned and the customer is notified by means of an appropriate exception.

Figure 33: **Place order use case part 1 – place order customer side**
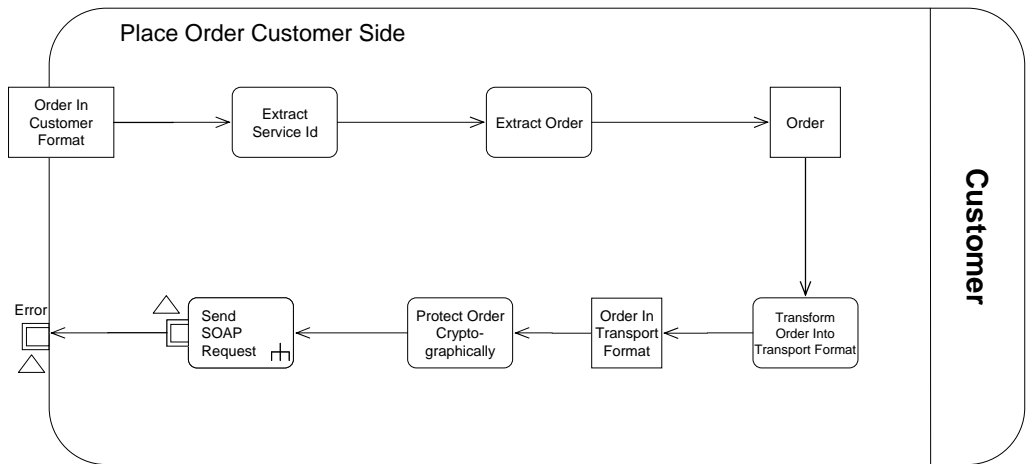
As soon as the platform server receives a place order SOAP request the server-sided processing of the transaction, displayed in the two diagrams below, starts. The actions following the "Extract Order From Transport Format" action are combined in a separate activity. The reason for this is the need to make the "Order" object available to the actions within the activity, since the order's identification number is required to retrieve the right order-object from the datastore node later on.

In the diagram below there are two points worth mentioning. First of all concerning the further processing of the order after the platform server has been given the go-ahead for dispatching the product orders to the suppliers. The entire order containing the different orders targeted at different suppliers therefore has to be retrieved from the oracle database. The different products are now sorted by the "Generate Partial Orders" action according to the suppliers they are provided by. The result is a collection of partial orders, which have to be sent separately to the different suppliers.

The second point concerns the handling of the exception that may be thrown by the "Send SOAP Request" activity. Exceptions of this type are issued by the "Send SOAP Request" activity via a pin with the isException attribute set. The including activity and the "Place Order Server Side" activity on the other hand use an activity parameter node with streaming capabilities and without the isException attribute for forwarding the exception. In short a pin or an activity parameter node with streaming capabilities is able to accept tokens also during execution. Only this streaming/isException construction allows the issuing of an exception during the processing of one partial order without automatically aborting the processing of the other partial orders.

Figure 35:   **Place order use case part 3 – place order server side**
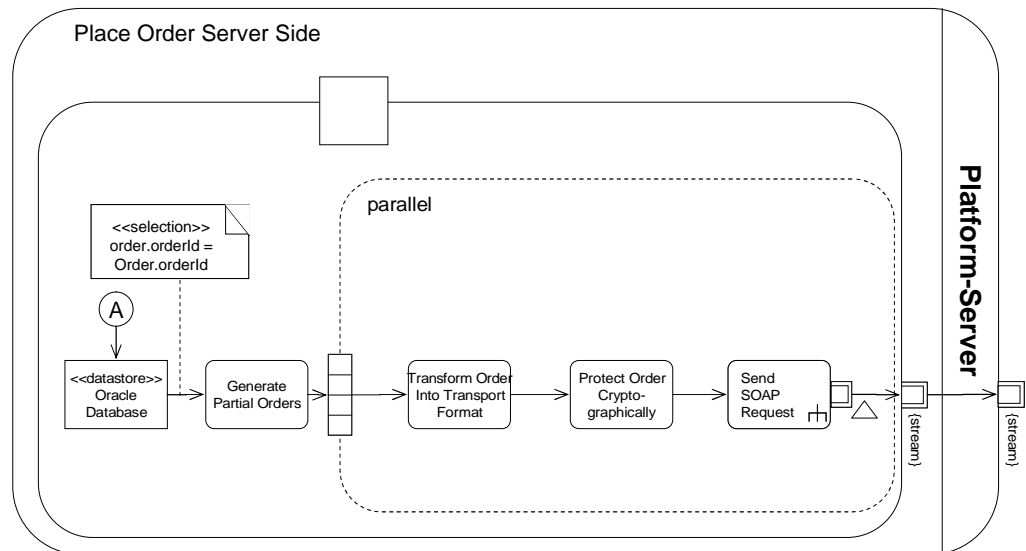
The supplier-sided processing of the transaction illustrated below starts as soon as the supplier receives a respective SOAP request through the "Receive Partial Order Delivery SOAP Request" accept event action.

**Figure 36:**   **Place order use case part 4 – deliver order**
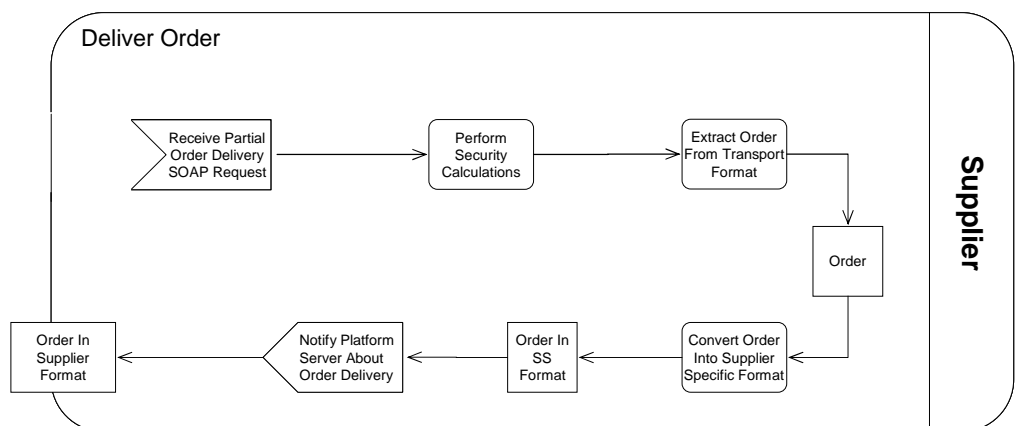
The diagram below displays the dispatching of a SOAP message. Even though there are actually two different types of SOAP transactions (those sent from the customer to the platform server and from the platform server to the supplier), they differ only in the name of their signaling actions and have thus been combined in one diagram.

The sender tries five times to dispatch a SOAP message. After five unsuccessful attempts the sender gives up and an exception is thrown.

After having successfully dispatched a SOAP message the sender waits until he receives a corresponding return message.

Place order use case part 5 – send SOAP request
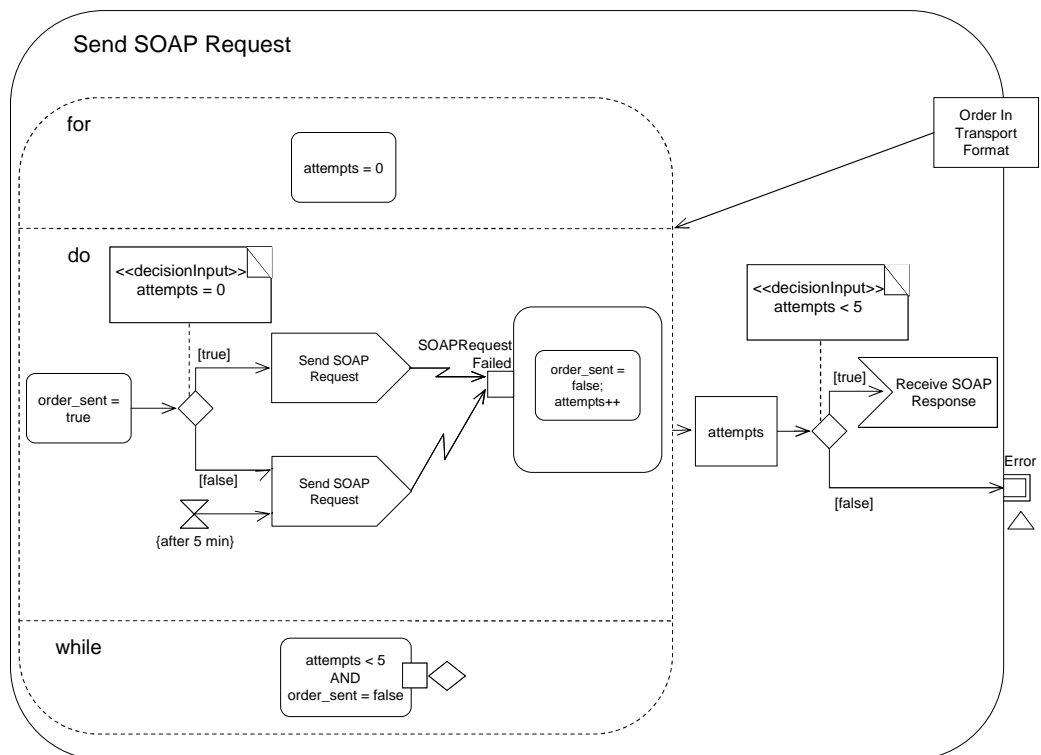
2.4   Conclusion

This section shall sketch the evaluation of activity diagrams according to a preliminary list of assessment criteria. The elaboration of a complete list of assessment criteria as well as a proper evaluation of the potential process modeling languages will be part of the next phase of the project.

One of the criteria regarded so far refers to the expressiveness of the considered languages. One way for measuring the expressiveness of a process modeling language are so called workflow patterns [15] that identify a cata-

40

log of typical recurrent process structures. Activity diagrams are capable of representing any of these workflow patterns. Moreover activity diagrams can express exceptions and interrupts and they can react to external events. Activity diagrams explicitly display the flow of control as well as the flow of data within a process. Additionally activities can be nested arbitrarily.

Another issue that will matter for the selection of an appropriate process modeling language refers to the unambiguousness of the language. According to the current standard of knowledge it can be made a note of that activity diagrams dispose of a clear notation while the description of the semantics of the elements doesn't always appear to be sufficiently unambiguous for a direct mapping to source code. However, a detailed investigation of the accuracy of the activity diagram specification is still an open task.

The analyzability of the semantics and syntax of an activity diagram also forms an assessment criterion. Syntax analysis allows for the verification of the syntax of a diagram while semantic analysis allows for the examination whether a diagram violates the semantics of the language specification. Activity diagrams still have to be examined with respect to these two properties.

Concerning usability activity diagrams dispose of a large number of elements, which complicates the acquisition of activity diagrams. Nevertheless some of the workflow patterns according to which the expressiveness of a process modeling language can be evaluated, can be represented only with considerable effort. Being expanded, activity diagrams get easily cumbersome. Another drawback of UML 2.0 activity diagrams is currently a lack of supporting tools.

An advantage of activity diagrams is their propagation. Being part of the UML standard it can be assumed that many people understand activity diagrams and that this will hold for the next years as well.

3   Petri Nets

In 1962 the mathematician Carl Adam Petri published his PhD thesis "Kommunikation mit Automaten" [11]. The purpose of this work was to generalize the existing automata theory by concurrency and parallelism concepts therefore proposing a new modeling language that comprises both a graphical representation as well as an algebraic mathematical one.

Finite automata (finite automata are automata with a limited number of internal states) had already been developed by Huffmann, Moore and Mealy between 1954 and 1956. Automata in general are used in computer science as mathematical models for representing information processing systems behavior [1].

With the publication of Petri's thesis it was the first time a general theory for distributed discrete systems was formulated. Until the mid 1980ies Petri's theory, subsequently known as "Petri nets", was mainly regarded by theorists. From then on, the "classical" Petri nets were extended by various features and several Petri net tools were published thereby considerably enhancing the practical use of Petri nets.

Nowadays the area of applications in which Petri nets are used is extremely wide. Petri nets are used in telecommunications for designing and simulating protocols; they are used for designing distributed software systems, for modeling (business) processes, for performance analysis, in production engineering, etc. A myriad of books and papers about Petri nets have been published; this great popularity can mainly be attributed to the fact that Petri nets combine a graphical representation and a strong mathematical basis that allows for the execution, simulation and analysis of Petri nets.

The following brief Petri net introduction will first of all introduce the classical Petri net and its components. These components or elements will then be used to formulate the most frequently used Petri net routing concepts. The subsequent chapter deals with the concept of triggering transitions that have been added to classical Petri nets more recently. The introduction of triggering is required since sometimes it is not sufficient for a task to be carried out that the containing Petri net is in the right state, but additionally certain events or initiatives by some actors external to the modeled system have to occur first. These additional prerequisites for a task execution are modeled by means of triggers. The subsequent chapter introduces the color, time, and hierarchical extensions that were essential for the practical suitability of Petri nets. A major example completes the Petri net introduction.

3.1 Basic Concepts

In general Petri nets consist of places and transitions. Directed arcs interlink places and transitions. Arcs may never connect two places or two transitions. Places are represented by circles while transitions are shown as rectangles. Connectors are depicted as arrows. Transitions have input and output places. The input places of a transition are the places at the sources of its incoming arcs. Accordingly the output places of a transition are located at the end of its outgoing arcs. In the classical Petri net, each place can hold at most one token; token are represented by black dots. While the structure of the Petri net is fixed the tokens may change their position. The current distribution of the tokens among the places of the Petri net determines its state.

A token passes from one place to its successor in the direction given by the arcs as soon as the intervening transition fires. A transition may fire if it is enabled. This is the case when there is one token at any of its input places. If a transition fires it consumes one token from each of its input places and produces one token for each of its output places. The firing of a transition results in the Petri net changing its state. Tokens traversing a transition via multiple incoming edges are merged. On the other hand if a place disposes of several outgoing edges a token will only be passed to one subsequent transition. To which one of the transitions the token will be forwarded is chosen non-deterministically. Several arcs entering a place form an OR-join. The following chapter will describe the routing concepts for Petri nets more in detail.

Because transitions can change the state of a Petri net, they are active components, which typically represent an event, an operation, a transformation, or transportation. On the other hand places are passive components, which stand for a medium, buffer, geographical location, state, phase or condition. Tokens are used to represent physical or data objects.

Extensions to the basic Petri net theory can be classified into three levels [13]:

- Level 1 Petri net systems are characterized by allowing only unstructured tokens, that is to say the places may hold at most one unstructured token at a time.The most important level 1 Petri nets are condition/event systems.

- Level 2 Petri nets, which are also referred to as place/transition Petri nets, have so called integer tokens, i.e. the places can serve as a counter by being able to hold several (optionally up to a certain limit) unstructured tokens at the same time. This feature can be used in combination with arc weights, which are available in level 2 Petri nets as well.

- Level 3 Petri net systems allow the use of high-level tokens, which are structured tokens with information attached to them. The most important group of level 3 Petri nets constitutes the group of high-level Petri nets that comprise colored Petri nets among others.

The example below taken from [14] illustrates fundamental concepts of Petri nets. It represents two traffic lights at a crossroad. The diagram becomes a bit more complex since it has to be made sure that one traffic light is always red. This is reached through the additional place "x" in the middle. Since "x" has two outgoing edges, either the transition "rg1" or "rg2" will fire first. Transition "rg1" and "rg2" both are enabled, because any of their predecessor places has a token.

Figure 38:   **Two sets of traffic lights – state 1**
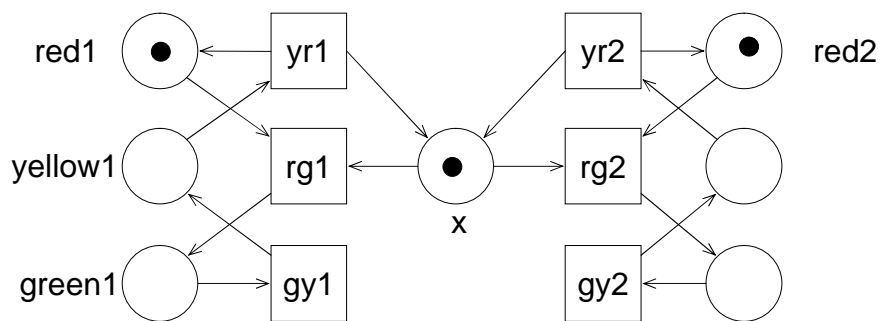
Let's assume that transaction "rg1" fires first. Tokens are consumed from place "red1" and "x". They are merged and placed on place "green1". The figure below shows how the Petri net looks like after transition "rg1" has fired.

Two sets of traffic lights – state 2
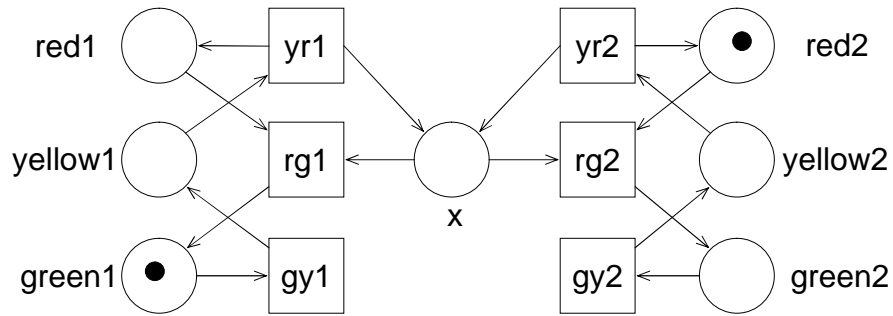
Now transition "gy1" is enabled and will fire. The figure below shows the corresponding Petri net.

Figure 40: **Two sets of traffic lights – state 3**
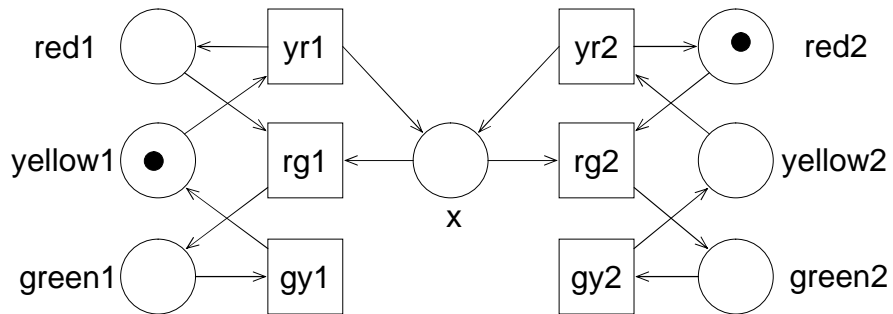
Because any predecessor place of transition "yr1" now has a token, it is enabled and ready to fire. Upon firing transition "yr1" duplicates the incoming token and places one duplicate on any of its successor places, i.e. place "red1" and "x". The figure below, which corresponds exactly to the initial state, shows the Petri net after transition "yr1" has fired. Notice that now either traffic light 1 or traffic light 2 can turn green, since both are enabled. Due to this conflict, the Petri net does not represent a fair behavior where the traffic lights turn green alternately.

Figure 41:   **Two sets of traffic lights – state 4**
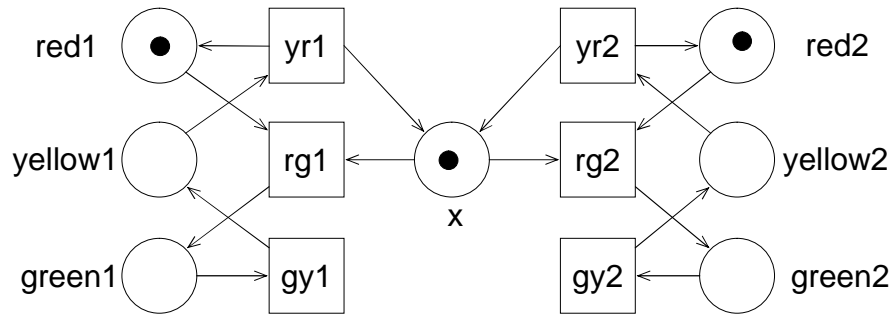
There may be multiple tokens present in one Petri net at the same time. Moreover the different tokens may either belong to the same task being handled by the process represented by the Petri net or they may belong to different tasks. Since the interaction of tokens from different tasks may be undesired, it must be made sure that they are handled separately. One way of uncoupling the tokens is to process the tokens that belong to the same task each in a separate instance of the Petri net. Another approach for separating tasks is only feasible in Petri nets with color extension, which will be introduced in detail later. The tokens are here identified as belonging to the same task by means of their color. Preconditions assigned to the transitions of the Petri net then make sure that only tokens belonging to the same task will be processed concurrently.

3.2   Routing Concepts

As indicated above the transitions within a process do not necessarily always have to be performed one after the other. Likewise the execution of a transition can be optional, two transitions may be executed concurrently or the same transition may be executed several times. Any of these routing concepts called AND-split, AND-join, OR-split, OR-join, and iterative execution of a transition can be resolved to the basic Petri net elements. However, in order to simplify the Petri net modeling additional transitions for routing purposes have been introduced for workflow nets [14]. Since the only aim of these elements is to illustrate the routing within a Petri net, they are also referred to as management tasks. Notice that these constructs can be mapped to standard Petri nets.

Whenever it is not determined whether two or more transitions are executed concurrently or in an arbitrary order one after the other, it's about parallel execution of transitions. The parallel execution of transitions is preluded by an AND-split. An AND-split transition produces a token on any of its succes-

sor places. An AND-join on the other hand is used to resynchronize parallel flows. An AND-join is only activated whenever there's a token on any of its input places.

Figure 42 introduces the specialized notation for an AND-split and AND-join. Here and in the following figures introducing the notation for routing elements the left side will always show the shorthand notation, while the right side will depict the semantically identical construction using the classical Petri net modeling elements.

Figure 42:   **Extended Petri net notation for AND-splits and -joins**
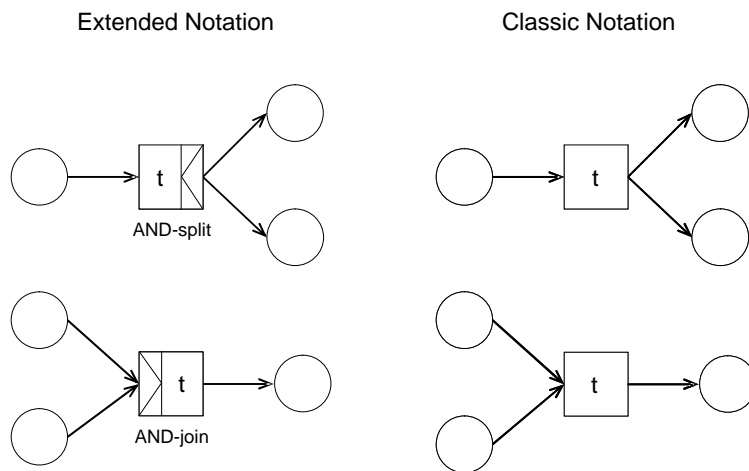
Figure 43 gives an example for using the routing elements AND-split and AND-join. Whenever possible the examples (like this one) correspond to the respective examples within the activity diagram chapter.

Figure 43:   **Example for an AND-split and -join**
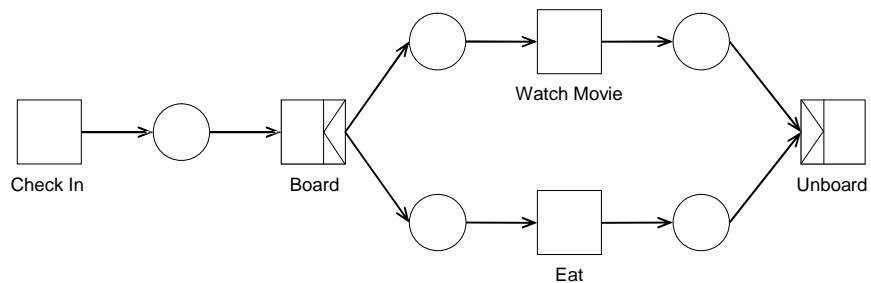
It's a matter of selective routing if only one of a number of alternative transitions shall be executed to the exclusion of the others. Two alternative flows begin with an OR-split, which forwards a token to only one of its outgoing edges. In classic Petri nets the decision which one of the alternative transitions shall fire is nondeterministic while in the case of Petri nets with color extension (which will be introduced in more detail later) the transitions may dispose of preconditions that determine on the basis of the "color" of the token which branch may be taken. Alternative flows are reunified by means of an OR-join. Even though there is a special element dedicated for modeling OR-joins an OR-join can also be represented by a place with several incoming edges. Since this representation is semantically equivalent it is the preferable way of modeling OR-joins as it reduces the number of elements in the diagram.

Figure 44 illustrates the notations for an OR-split and OR-join and the corresponding routing mechanisms realized by means of the classic Petri net elements.

Figure 44: **Extended Petri net notation for OR-splits and -joins**
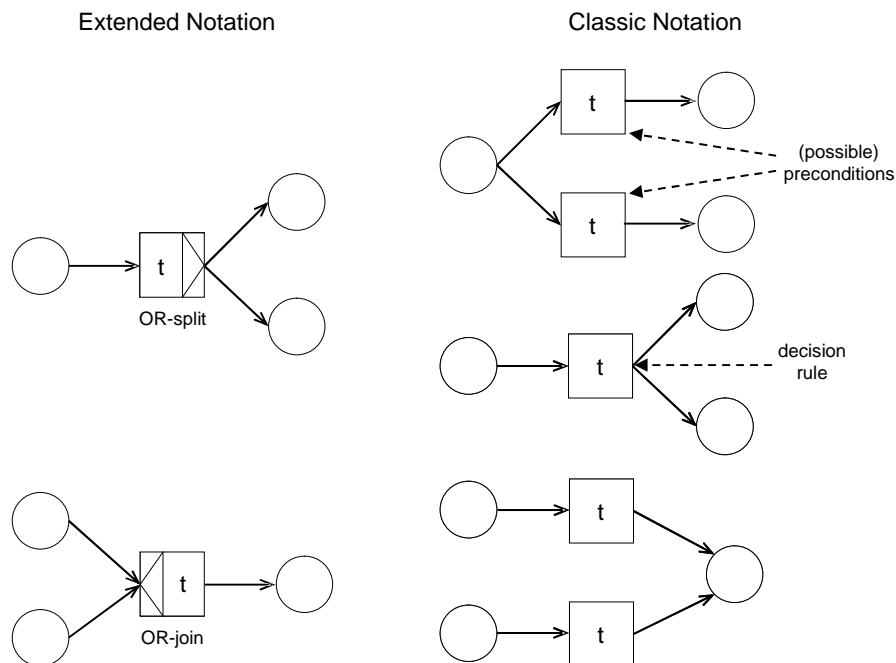
An example for a Petri net comprising an OR-split and OR-join is depicted in Figure 45. This example represents the Petri net version of the example given in Figure 15.
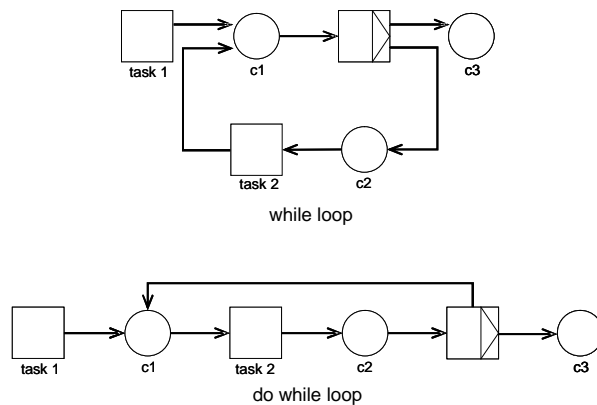
Iterative routing constitutes another routing mechanism. It refers to the case in which one transition is executed repeatedly. Both "while" and "do while" loops can be modeled quite easily using the elements introduced so far.

Figure 46 illustrates how both while and do while loops can be modeled in Petri nets.

Figure 46:   **Modeling "while" and "do while" loops**

3.3   Triggering of Transitions

Like already mentioned in the last chapter a transition has to be enabled in order to fire, i.e. the containing Petri net has to be in the right state. However, considering a Petri net as the model of a process, the execution of a subtask represented by a transition may depend on some additional external prerequisites, such as an employee that is involved in the processing of the subtask has to apply explicitly to the task before the processing can start. These additional external prerequisites that have to be fulfilled for a subtask to start (provided that it has been enabled) are called triggers.

Three different types of triggers can be distinguished:

1. A so called resource initiative, such as an employee that starts applying to a task

2. An external event like the arrival of an EDI message

3. A time event

Figure 47 sets an example for the utilization of the three different trigger types. Let us assume that incoming orders shall be processed by a responsible official. For the processing of the order to start an order first has to arrive. This prerequisite (external event trigger) is indicated by the small envelope symbol above the "process order" transition. Moreover, after the order has arrived, the person in charge of the processing of the orders explicitly has to turn towards the order and start processing it. This resource initiative trigger is indicated by the small downward facing arrow besides the envelope symbol. After the order has been processed, it shall be delivered to the customer. But since products are sent to the customer only once per day, the delivery time has to be awaited for the delivery to start. This (time event) is indicated by the small clock symbol above the "deliver products" transition.

Figure 47:   **Example for the three trigger types**
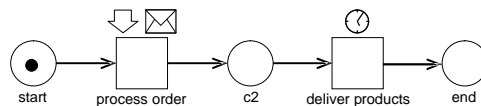
Triggers can also be modeled using the classic Petri net elements. Each trigger is then represented by an additional place connected with the transition that needs a trigger in order to fire. The appearance of a token on this additional place now corresponds to the triggering of the task.

3.4   Higher Petri Nets

Petri nets have been extended in many ways in order to be able to model complex situations in a structured and more accessible way. The three most important extensions are the color extension, the time extension and the hierarchical extension. Petri nets with these extensions are called high-level Petri nets.

3.4.1 Color Extension

In classical Petri nets it is impossible to distinguish between two tokens in the same place. This shortcoming is addressed by the color feature, which allows for the coupling of the characteristics of an object with the corresponding token. The color extension makes it possible to provide each token with an arbitrary color or values.

In contrast to the classic Petri net the number and values of the tokens produced by a transition here may depend on the number and values of the consumed tokens. In contrast to the classic Petri net not all of the output places automatically receive tokens upon firing of a transition. Now a choice is made according to the information available on the token, which place will receive a token.

In colored Petri nets it is also possible to set preconditions as a necessary prerequisite for a transition to become enabled. Not only the tokens then have to be available at the respective input places, but also the preconditions have to be fulfilled for the transition to become ready to fire. Thus preconditions act like transition guards. Whether the precondition of a transition is fulfilled or not depends on the values of the tokens to be consumed.

In contrast to classic Petri nets the graphic representation of colored Petri nets now doesn't contain the entire information any more. In addition to the diagram it has to be specified which preconditions there are for which transitions, how many tokens are produced for which output place by a transition upon firing, and the values of the tokens produced, whereas the number and values of the tokens produced by a transition may depend on the values of the tokens consumed.

Figure 48 shows an example of how colored tokens can augment the expressiveness of Petri net models. The example taken from [14] depicts a process for handling technical faults in a product department. First of all a fault is examined and categorized based on the token attributes that hold the relevant properties of the fault (like e.g. a description of the fault, the identity of the broken component, etc.). Upon categorization the fault can sometimes be solved right away. In order to exclusively determine whether a token shall proceed to the end place "solved" or to the place "needs repair" the "categorize" transition disposes of a decision rule that specifies under which conditions the token shall proceed to which place. In case the fault couldn't be fixed immediately it will first now be repaired in a separate step and then be tested. The "test" transition again disposes of a decision rule that determines based on the properties of the token whether the fault can be considered to be solved, whether further repairing is required or whether the fault is irreparable and the affected component needs to be replaced.
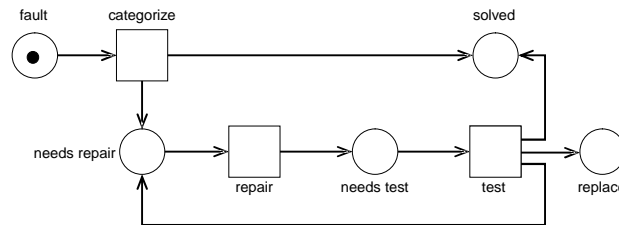
3.4.2 Time Extension

The time extension allows for the association of time information with tokens and transitions. This time information denotes from which time on a token is available for being consumed by a transition. Hence a transition will only be enabled at a point of time equal to or bigger than the time denoted by the required input tokens.

If there are several transitions ready to fire the one enabled by the tokens with the earliest timestamp is the one that fires first. If there are several transitions enabled by tokens with the same timestamp a non-deterministic choice is made. Concerning the consumption of tokens, tokens are consumed on a FIFO basis, i.e. the token with the earliest timestamp is the one to be consumed first.

Transitions may have a certain delay. The transition delay doesn't actually affect the time the transition needs for processing tokens, but only results in the tokens being issued to receive a new timestamp that corresponds to the time they have been consumed plus the transition delay. The delay of a transition may also depend on the values of the processed tokens. Likewise a transition may have a fix or a random delay.

Figure 49 below shows the exemplary Petri net from Figure 38 - Figure 41 expanded by time data that specifies the processing time of every transition (indicated by the small numbers within the rectangles attached to the arcs). The figure also displays the timestamps of the tokens (all zero) at the beginning of the processing. Assuming "rg1" fires the two tokens from place "red1" and "x" will be merged and the resulting token will be placed on place "green1" receiving a new timestamp with the value 25. If now "gy1" fires a token with the timestamp 30 will be produced on place "yellow1", etc.
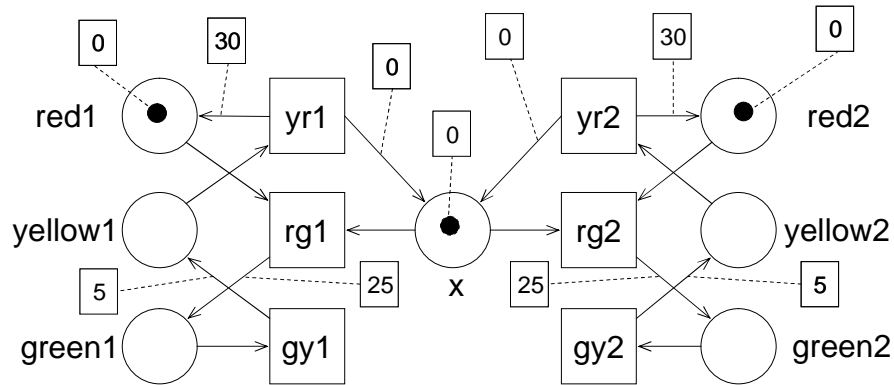
3.4.3 Hierarchical Extension

The hierarchical extension allows for the better structuring of a Petri net. Therefore a new element is introduced, a so-called process. A process represents a subnet of a Petri net comprising places, transitions, and arcs. Moreover processes again may contain sub-processes thus leading to a hierarchically nested process structure. A process occurs in two different forms: as a double-bordered square within a process and as the definition of a process showing the places, transitions, arcs, and sub-processes it consists of.

Figure 50 shows the two representations of a process. With "repair" the example from Figure 48 this time contains a structured transition that comprises several processing steps as denoted in the process definition spanned by the dashed line at the bottom of the picture.
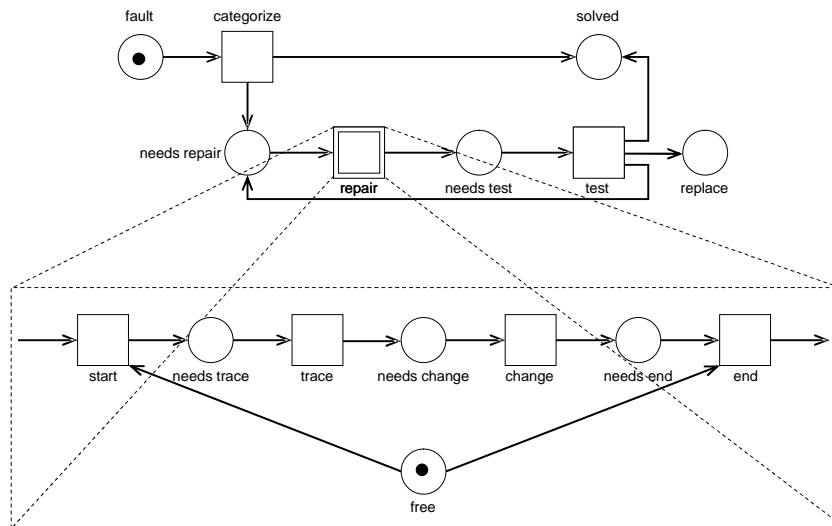
3.5   Example

The following pictures show the same E-commerce platform introduced already in the Activity Diagram chapter this time modeled as a Petri net.

Figure 51 below shows the customer sided processing of an order and the first part of the server sided processing from the reception of the "Place Order" request up to the storage of the order in the database. The "send_soap_request" transition is a process in the abstract representation hiding the contained elements. Due to spatial constraints the Petri net had to be divided into several diagrams. The small stars with the index numbers shall indicate where the token flow continues.
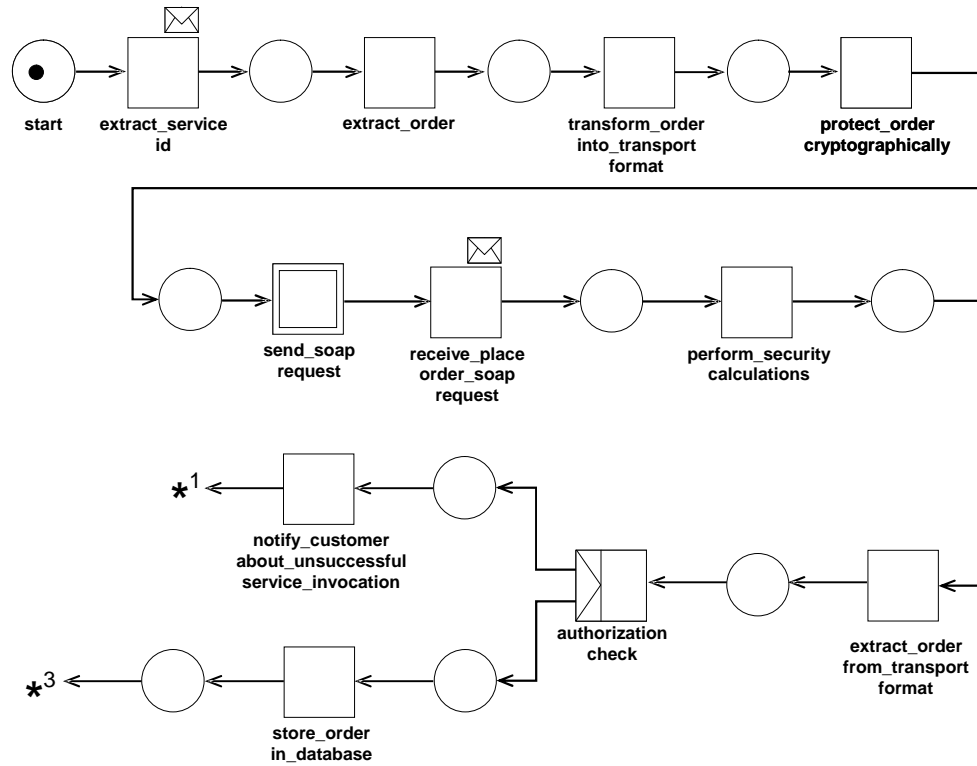
Figure 52 shows the definition of the "send_soap_request" transition.

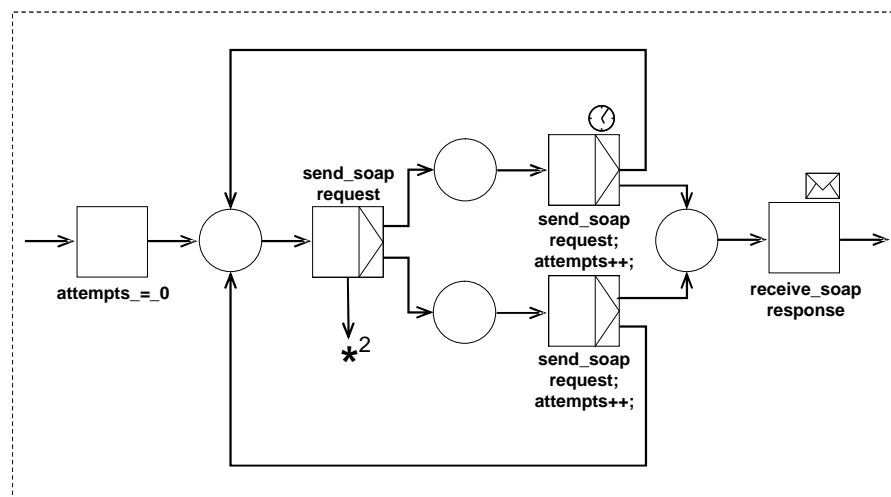**Figure 52:** **Place order use case part 2 – Petri net version**

56

Figure 53 shows the further processing of an order at the platform-server comprising optionally the waiting for an order approval by the superordinated customer, the notification of the customer that the order has been placed successfully, the retrieval of the order from the database and subsequently the generation of partial orders that will be sent to the different suppliers.

Figure 53:   **Place order use case part 3 – Petri net version**
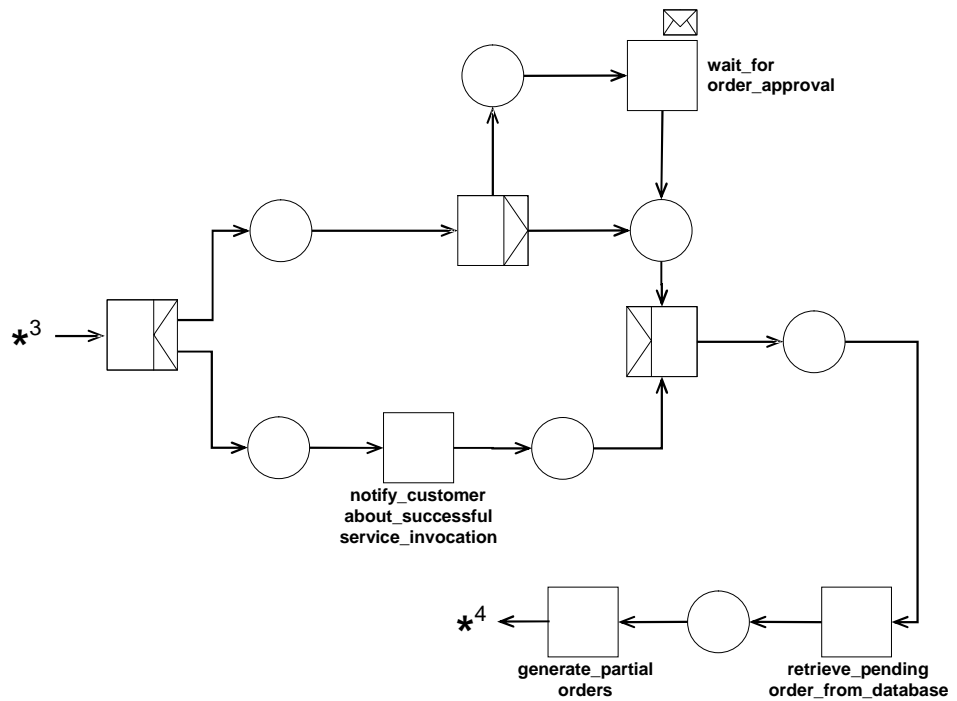
In Figure 54 the last processing steps at the e-commerce server are denoted, which comprise the transformation of every partial order into the transport format followed by the delivery of the order by means of a SOAP request. Moreover it depicts how the partial orders received by the e-commerce model are delivered to the supplier's ERP system.

Figure 54: **Place order use case part 4 – Petri net version**
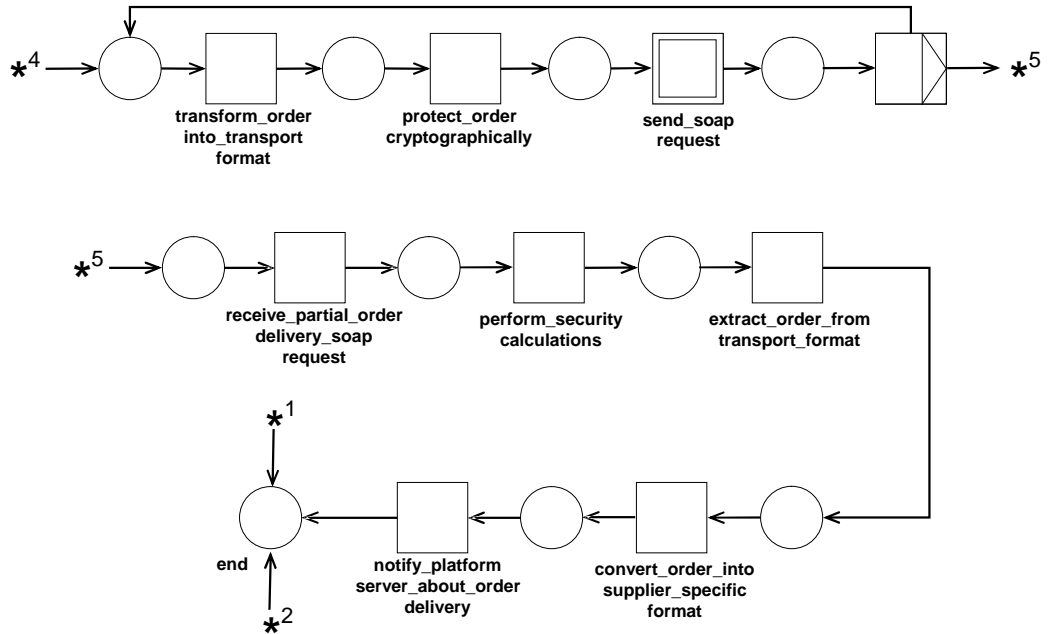
3.6   Conclusion

As indicated above, Petri nets have been successfully used in different application domains to model the behavior of dynamic systems with a static structure. While the classical Petri net provides a limited set of constructs with well-defined semantics, modeling real world processes with classical Petri nets typically results in Petri nets of considerable size. For this reason, higher-level Petri nets have been introduced. Higher Petri nets allow rather compact modeling of complex processes at the cost of fewer possibilities to analyze processes. In the context of workflow nets, tools for analyzing workflow process specifications have been developed that exhibit efficient run-time behavior on a large subset of workflow nets. The strengths of Petri nets in general and workflow nets in particular are their formal foundation and graphical representation. However, many people feel that even these higher-level Petri nets are not easily understandable for non-experts that inhibit even broader applicability of this important process modeling approach.

4　Business Process Modeling Notation

The Business Process Modeling Notation (BPMN) [6] allows the definition of business processes in diagrammatic form. The notation has recently been proposed by the Business Process Management Initiative (http://www.bpmi.org), whose members are major well-known software vendors.

In the presence of many different process modeling approaches the question arises, why yet another one? The members of the Business Process Management Initiative yield expertise and experience in many existing notations and combined the best ideas into one single standard notation. This standard should be easily understandable by different business users. Therefore simplicity was a goal, which stands in contrast to another goal, the visualization and direct mapping of XML languages designed for the execution of business processes.

The BPMN is designed to bridge the gap between business process design and implementation. The long term aim is to automatically visualize business process execution language programs into standardized business process diagrams and vice versa. The effort of a human translator is then no longer needed. The BPMN draft specification already defines a mapping from BPMN to BPEL4WS (Business Process Execution Language for Web Services) [2], as will be described later in this chapter.

The notation is intended to be used with business processes on three different levels of abstraction. This includes private, abstract and collaboration processes. The first is generally known as workflow, it describes business processes, which are internal to a specific organization. Abstract (public) processes represent the communication between a private business process and another process or participant. They contain only the activities needed for communication. Collaborations define a sequence of activities that represent the message exchange pattern between two or more business entities.

4.1　Notation

This section describes the achievement of the above goals, especially the dissolving of the requirement simplicity and the potentially conflicting requirement of mapping complex business processes to business process execution languages.

Simplicity is achieved through a basic set of core elements for grouping, primary modeling and connectivity. Using only these core elements, many

business processes can be visualized. The core elements can be enhanced based on their basic visual notation. This means, adding something to the visualization of a core element, does not change the basic shape. This way, even new and unknown elements can be mapped back to core elements.

To reach the mapping to business process execution languages, more information is needed. This information is provided in form of attributes for each element. Attributes can be set automatically by tools; this includes unique element ids, default names or conditions. Some default values are defined by the BPMN specification, always specifying the core format of the element. By modifying those attributes, enhanced and complex variations can be created. However, it is not necessarily needed to change the attributes to receive simple behavior.

Beginning with the core elements, the notation of BPMN is introduced in the following subsections. This is just an outline of the specified elements; the complete reference can be found in the BPMN specification [6]. A real-world example follows in section Example.

Figure 55:   **BMPN core elements**
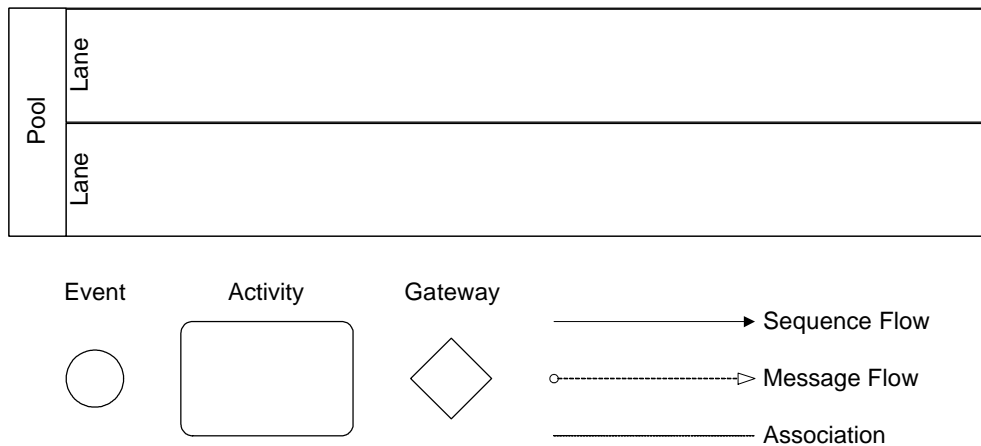
4.1.1   Core Elements

The BPMN core element set can be divided into three groups. These groups include elements for grouping, primary modeling and connectivity. Everything in BMPN can be modeled by further specializing single core elements. The core elements are shown in Figure 55.

The primary modeling elements are events, activities and gateways. Events are something that "happens" in the course of a business process. They af-

fect the flow of the process and can have a trigger or a result. An activity is work a company performs; it can be atomic or complex. Gateways are used as controlling structures for sequence flows.

Sequence flows connect events, activities and gateways and therefore belong to the connectivity elements. The sequence flow defines the possible flow of the process. Message flows show the flow of messages between businesses. An association connects further information, like descriptions, to the core elements.

All primary modeling elements are placed inside pools. A pool is a container for grouping a set of activities for a particular organization. To further decompose an organization into specific units, so-called swim lanes inside a pool can be used. The pools can be black or white boxed. A black box pool hides it inside details; communication can only occur to the outside line of the pool, whereas a white box pool shows it inside details and allows communication to inside elements.

Figure 56:   **Example using BPMN core elements**
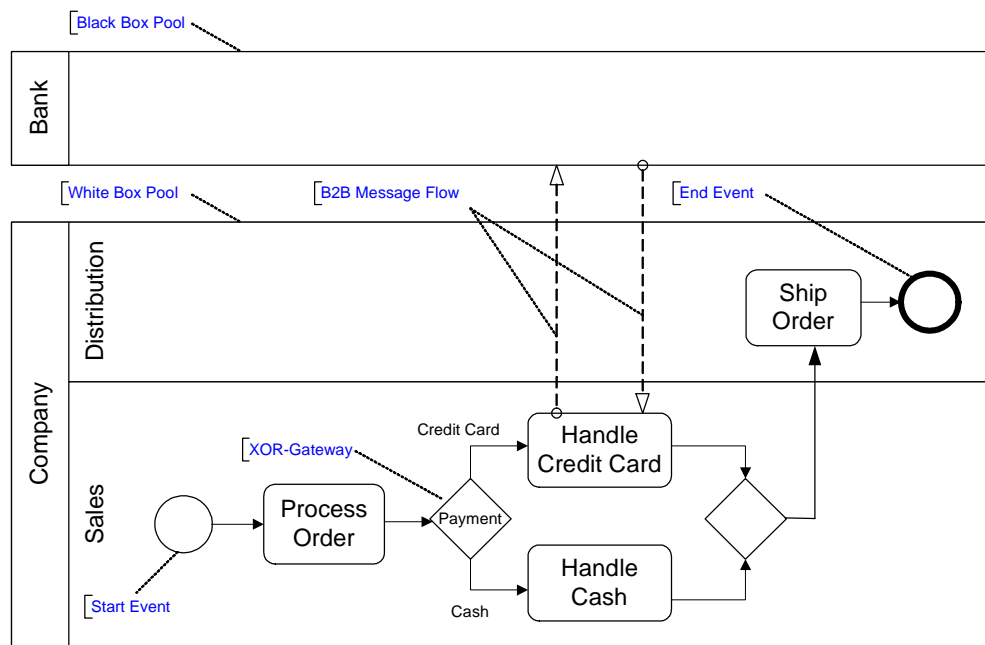
An example of a BPMN diagram using almost exclusively the core elements is given in Figure 56. That particular process begins with a generic event, called start event. After processing the order, a decision for a payment method must be made. If "Credit Card" is chosen, the transaction must be confirmed by another participant: a bank. The bank is a black box pool in this example; internal processes are unknown. Afterwards the order can be

61

shipped. The process ends with an end event, a slight modification of the basic event type.

4.1.2 Events

As explained earlier, the core elements can be modified to achieve more complex behavior. An example was given in Figure 56, where the core element event has been named start event and a derived shape, with a bolder outline, end event. The intermediate event is a third form, visualized using two circles. It does affect the flow of the process, but will not start or end it.

Figure 57:   **Complete list of BPMN event types**

Based on start, intermediate and end events, different types of events have been specified. They are shown in Figure 57. In BPMN all start events produce a "token", which follows the sequence flow of the process. All end events consume a token. This is in contrast to other notation like Petri nets, where transitions consume and produce tokens. The start and end events are optional. If they are suppressed, every activity that has no incoming sequence flow acts as a start event and every activity that has no outgoing sequence flow acts as an end event. The process finishes when all parallel paths have been completed.

Figure 58:   **Example of the message events**

The message event can be used to model communication with other participants, which can be omitted in the diagram or exist in another pool. The

62

message start event triggers the start of the process after the arrival of a message, while the message end event finishes the process and sends a message to a participant. The message intermediate event stops the process to wait for the receipt of a message. An example is shown in Figure 58.

Example of the timer events
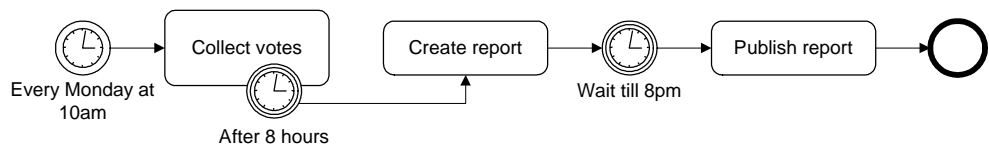
The timer start event triggers the start of a process to a specific time or time cycle. The timer intermediate event delays the process flow for a given time value or interrupt an activity at a specified time or duration. An example for a voting process is shown in Figure 59.

The exception events are used within exception handling; they are described in section 2.1 (Activities).

The cancel events are used to abort transactions, while the compensation events are used for compensation handling. The usage is described in section 2.1 (Activities).
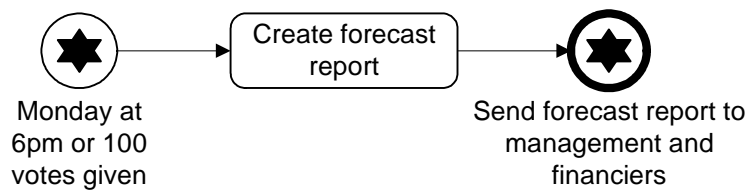
Figure 60:   **Example of the rule events**
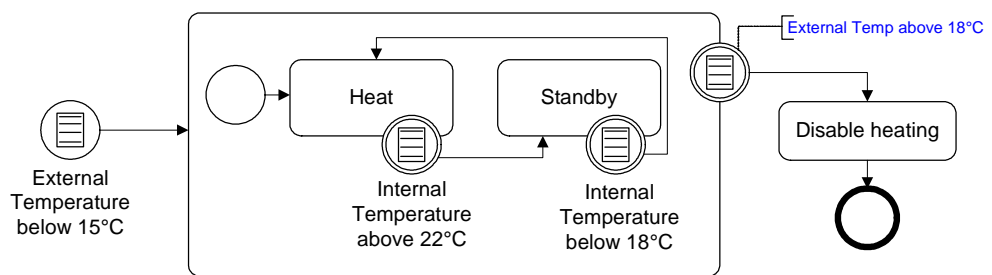
The rule start event works similar to the timer start event; the condition is set to a rule like "Temperature below 15°C". The rule intermediate event is used for exception handling when a named rule becomes true. An example is given in Figure 60. Because the rule intermediate event is only used for exception handling, it has to be placed on the edge of an activity, as it creates exceptional flow. Exceptional flow is equal to sequence flow; it just opens a separate path.

The link start and end events connect the end of one process to the start of another. The link intermediate events connect the end of one process to the start of an event-based exclusive decision. This type of decision is described in section 4.1.4 (Gateways). The final usage of the link events is still an open issue in the BPMN draft specification; therefore no concrete example can be given.

Example of the multiple events

Monday at
6pm or 100
votes given

Create forecast
report

Send forecast report to
management and
financiers

The multiple event is used if there exists different ways of starting, ending or disturbing a process. The multiple events can be seen as a list of almost any other events, automatically choosing the triggering event. An example is given in Figure 61.

The terminate event indicates a fatal error and therefore stops all (parallel) activities in the process without any exception or compensation handling.

4.1.3 Activities

Activities can be divided into processes, sub-processes and tasks. Sub-processes can contain other sub-processes. IN BPMN a process is activity performed within a company or organization. The term business process refers to one or more of those processes. Each process may have its own sub-processes and is contained within a pool. If the business process diagram contains only one pool, then the pool can be omitted.

Figure 62:   **BPMN sub-processes**
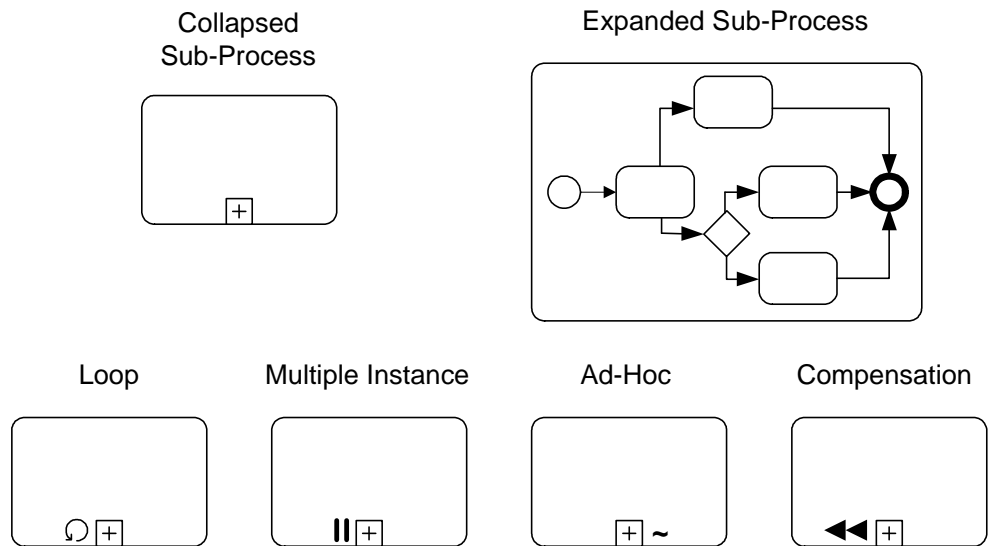
A sub-process defines compound activity. It can be shown collapsed, hiding its details or expanded, showing its inside details. A collapsed sub-Process is marked with a square and a plus sign inside (see Figure 62). Sub-processes can have four special markers for further specializing the type. Those markers can be combined in any combination with the exception of loop and multiple instances together.

The loop marker announces sequentially repeated activity, whereas a multiple instance marker denotes parallel activity. The ad-hoc marker defines a variable execution order of the contained activities. A compensation sub-process groups several activities for compensation.

Figure 63:   **BPMN transactions**
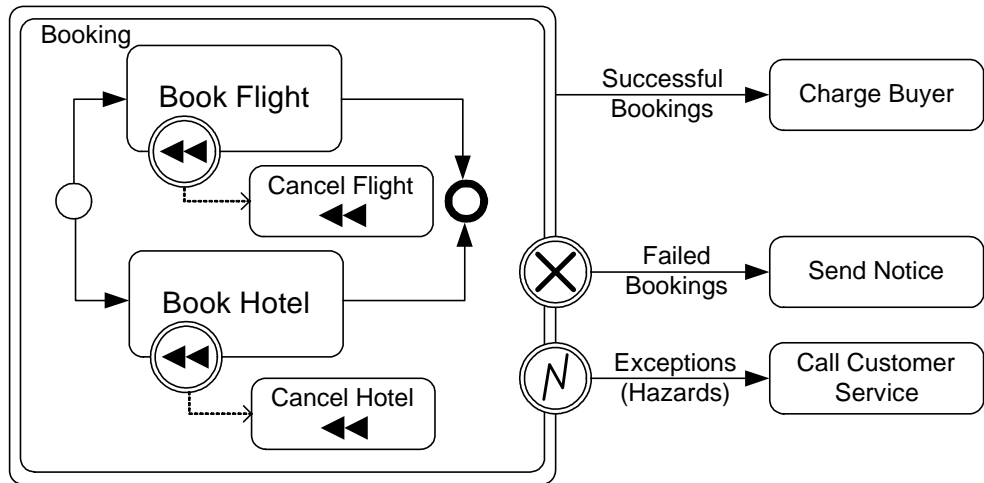
A special kind of sub-process is a transaction, using a double-lined bound-
ary (see Figure 63). Transactions can have three basic outcomes: success-
ful completion, failed completion and exceptions. Successful completion re-
sults in normal sequence flow. If the transaction is canceled, all intermediate
compensation events are called. After successful compensation handling,
the flow continues from the intermediate cancel event. If an exception oc-
curs and neither normal completion nor cancel is possible, the activity is in-
terrupted without compensation and the flow continues from the intermedi-
ate exception event.

**Figure 64:**   **BPMN task types**
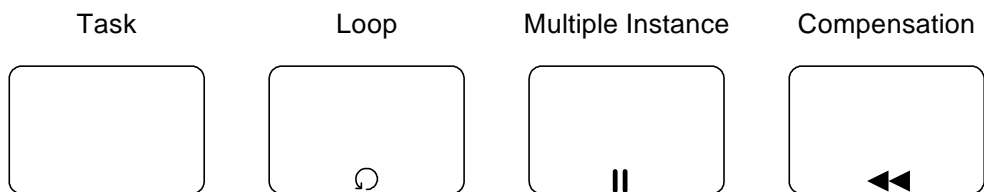
A task is an atomic activity within a process. In BPMN four different types of
tasks are defined (see Figure 64). A basic task is a single activity. A loop is
a sequentially repeating activity, whereas a multiple instance task denotes
parallel activity. The compensation task is used for compensation activity,
which is linked to a compensation event (see Figure 63).

4.1.4 Gateways

Gateways are used as controlling structures for sequence flow. They can decide, split or merge the flow of the process. The possible gateways are shown in Figure 65.

Figure 65: **BPMN gateway types**
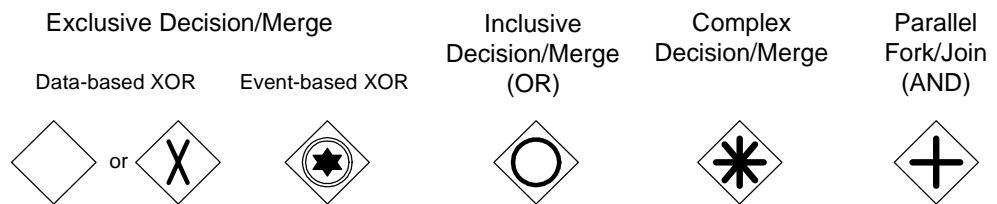
The interesting types are the event-based XOR and the complex gateway. The complex gateway defines conditions, which can refer to sequence flow and process data. With complex gateways, constructs of arbitrary other gateway types can be merged into one single gateway.

Figure 66: **BPMN event-based XOR gateway example**
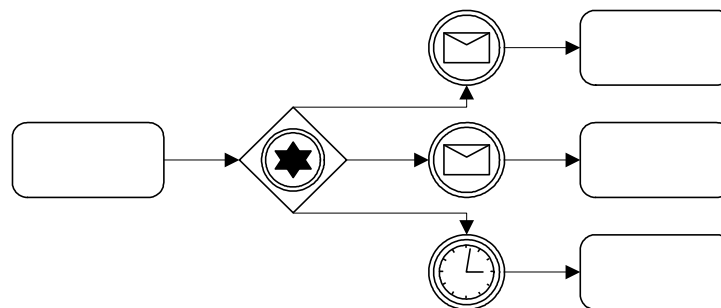
The event-based XOR gateway is used to model decisions based on events rather than process data. The process flow will continue if one of the specified events becomes true. In the example in Figure 66, this might be a "Yes" message, a "No" message or a timeout.

4.1.5 Sequence and Message Flow

Sequence flow occurs between activities in a process, whereas message flow occurs between different processes. As mentioned earlier, a process resists in a pool. Therefore sequence flow can only occur inside a pool; it can not cross the borders. This is the intention of message flows, representing B2B communication. The flow types are shown in Figure 67. Another example for message flow is shown in Figure 56.
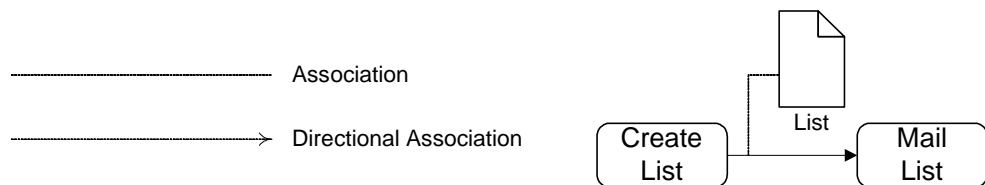
Figure 67:   **BPMN sequence and message flow types**

4.1.6 Associations

Associations connect information and artifacts with elements. The only artifact yet specified is a data object. Associations are shown in Figure 68.

Figure 68:   **BMPN association types**

4.1.7 Attributes

Every BPMN element owns attributes, which exactly define it. These attributes are used for mapping to executable languages. Almost every element has the basic attributes `Id`, `Name`, `Pool` (Lane), `Assign` and `Documentation`. The first three attributes can be set automatically by modeling tools, whereas `Assign` describes an expression that shall be evaluated when the flow of the process arrives at the specific element.

Most of the Attributes are fully or partially visualized in the graphical representation of the element. The loop marker of a sub-process for example, is backed by `LoopCondition`, `Counter`, `Maximum` and `TestTime` attributes. events usually have a `Trigger` attribute, which defines the type of the event. Based on the type, other attributes can be set, for example a `Timer`, an `Exception` or a `Rule` Attribute. Other elements have other attributes, which can be found in the official specification [6].

4.2 Example

In this section, the example introduced in the UML Activity Diagram part is visualized using BPMN. The notation allows different levels of abstraction; a good modeling start point is an abstract (or public) process view of all par-

68

ticipants. A first view of the E-Com platform could therefore show the different businesses and their message flows (see Figure 69).

**Figure 69:**   **High level message flow for an e-commerce platform**
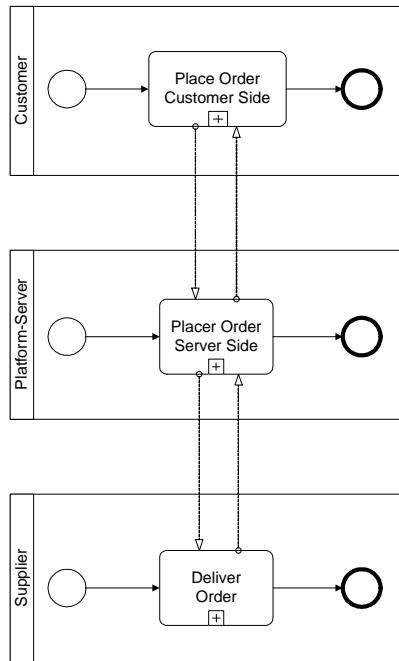
The platform consists of three participants, each represented by a pool. The pools are white boxed, showing its inner details. As can be seen by the small plus signs, all activities can be expanded. The first sub-process is `Place Order Customer Side` (see Figure 70).
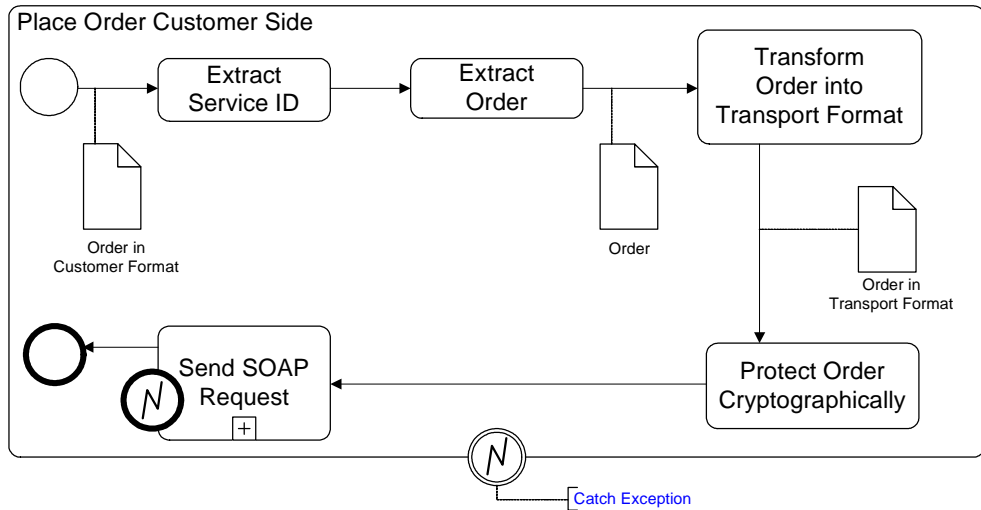
The sub-process shown only contains sequential activities. The flow of the data and the data types are represented as informal data object associations. The `Send SOAP Request` activity is itself a sub-process, capable of throwing an exception event. This event is captured at the border of the sub-process and can be further processed.

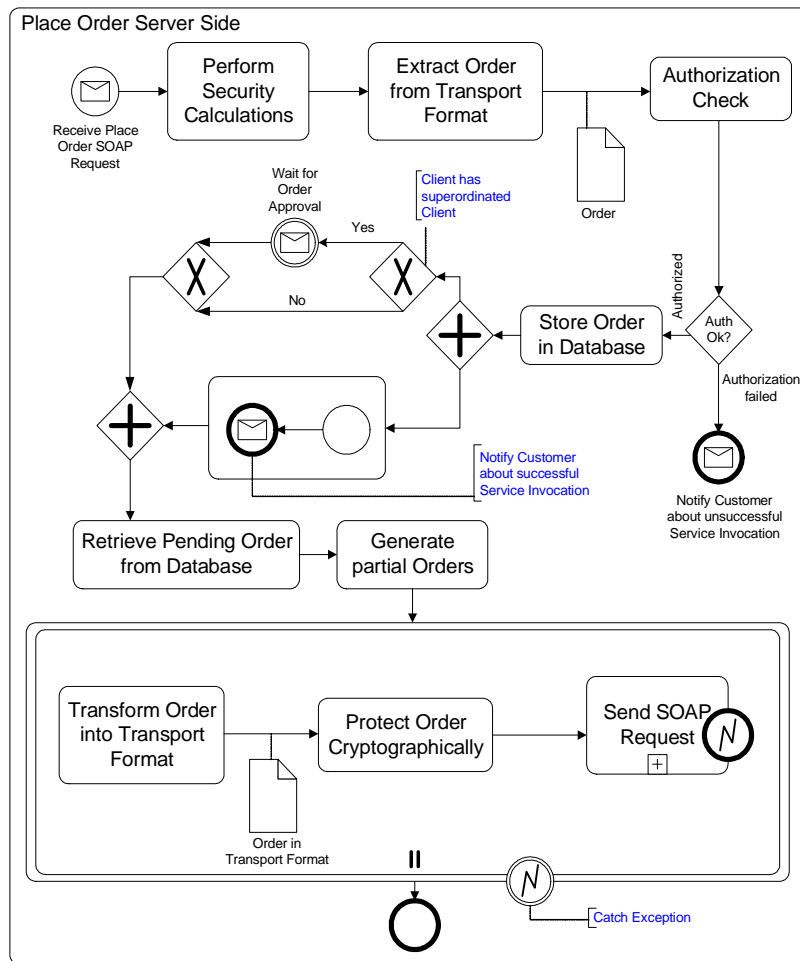**Figure 71:**   **Expanded sub-process "Send SOAP Request"**
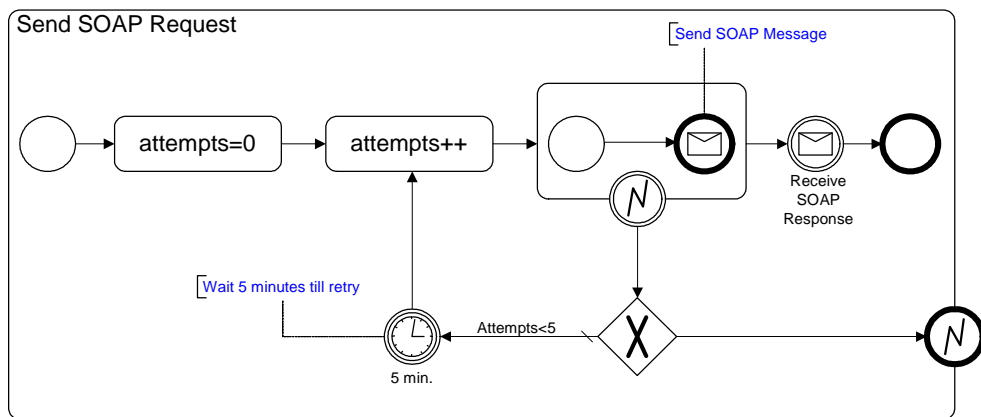
The `Send SOAP Request` sub-process shown in Figure 71 tries five times to send a SOAP message with a delay of five minutes for each retry. If the sending finally fails, an exception event is generated. If the message is sent successfully, the process stops until the arrival of the SOAP response. The expanded sub-process, which sends the SOAP message using a message event, is needed to catch the exception. The dispatching of a message could also be modeled using an outgoing message flow from the border of

an activity. It is than internally mapped back to the representation shown here.

The expanded `Place Order Server Side` sub-process is shown in Figure 72. The process starts with receiving a SOAP message from a participant. After checking the authorization, a successful/unsuccessful service invocation message is send back to the participant. If a superordinated client exists, the process stops until the arrival of an approval message. Afterward several parallel sub-processes are invoked, producing requests for partial orders send to other participants. These partial order requests are shown here as transactions, meaning all or nothing of each will be completed. The `Place Order Server Side` sub-process finishes when all parallel sub-processes have finished.

Figure 72:   **Expanded sub-process "Send Order Server Side"**

The `Deliver Order` Sub-Process (see Figure 73) is invoked as soon as it receives a partial order SOAP message. It performs some sequential activities and finally sends a SOAP response.

Expanded sub-process "Deliver Order"
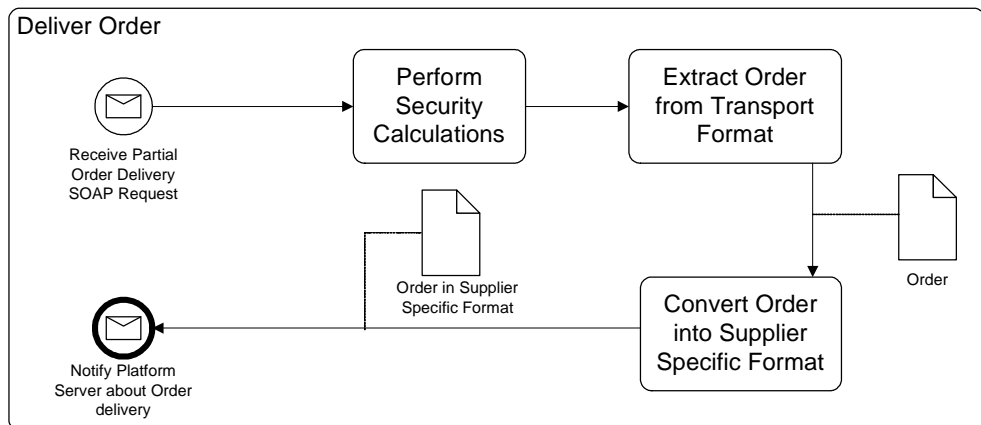
The Business Process Modeling Notation aims to be more "higher level" than UML; it contains no object flow or storage. The former can be displayed informal using associations, as can be seen by the data objects in the example diagrams. The later is displayed using activities. The notation uses the concept of message flows to represent B2B communication. It assumes that every process is placed inside a pool and that the "horizontal" communication line inside the pool consists of sequence flows between activities. The "vertical" communication line crosses pools and uses message flows for communication between different processes. Using these concepts, it is possible to model different levels of abstractions in a very simple but yet powerful way.

4.3 Mapping to executable languages

The mapping of BPMN to executable languages is still under development. An executable language allows the computerized execution of programs or instructions written in this language, either interpreted or compiled. The working draft specification [6] defines an incomplete mapping to BPEL4WS [2]. To understand the mapping from BPMN to BPEL4WS, a short introduction to BPEL4WS is given, followed by the basic mapping rules. To conclude the mapping section, a short example is given.

4.3.1 Mapping BPMN to BPEL4WS

The Business Process Execution Language for Web Services (BPEL4WS) deals with the orchestration of web services. In the context of BPMN, a web service can be seen as a way to perform an activity, as complex as ever it might be.

BPEL4WS is a XML language which defines a model and grammar for describing web service based business processes. It relies on other specifications like WSDL [16] for describing web services or XPath [17] for addressing elements. Enhanced features like fault and compensation handling are included.

BPEL4WS is a convergence of several older specifications. It is supported by a group of big players like BEA, IBM, Microsoft, SAP and Siebel and has a still emerging tool support, for example from Collaxa (http://www.collaxa.com).

Figure 74 shows the mapping of selected BPMN elements to BPEL4WS. Not every element is already specified. Every pool in BPMN maps to its own BPEL4WS document. Since BPMN allows the omitting of elements like start and end events as well as the creation of arbitrary circles, the diagram has to be analyzed. Therefore the BPMN specification mentions the concept of "Token Analysis", using a token to traverse all possible sequence flows of the process. The details are yet an open issue.

Figure 74:   **Mapping selected BPMN elements to BPEL4WS**

BPMN element		BPEL4WS element(s)
-	Assign attribute for every element	`<assign>`
✉ ➔	Message Start Event, Link Start Event	`<receive>`
✉	Message End Event	`<reply>`
N	Exception End Event	`<throw>`
◄◄	Compensation End Event	`<compensate>`
➔	Link End Event	`<invoke>`
⦿	Terminate End Event	`<terminate>`
✉	Message Intermediate Event	`<receive>`, `<on-Message>`
⏱	Timer Intermediate Event	`<wait>`, `<throw>`
N	Exception Intermediate Event	`<catch>`
◄◄	Compensation Intermediate Event	`<compensationHan-dler>`

BPMN element		BPEL4WS element(s)
	Sub-Process	`<invoke>, <re-ceive>, <reply>`
	Task	`<receive>, <re-ply>, <invoke>, <while>`
	Data-based Gateways	`<switch>`
	Event-based Gateways	`<pick>`

Almost every BPMN element has the `Assign` attribute, which is mapped to the corresponding BPEL4WS `<assign>` tag. As can be seen in Figure 74, most events have a one to one mapping to BPEL4WS. Exceptions are the timer and message intermediate events, which could map to different tags, depending on their configurations. In normal sequence flow, they are mapped to a `<receive>` or `<wait>` tag. If they are attached at the border of an activity, the message intermediate event is mapped to the `<onMessage>` tag, whereas the timer intermediate event maps to a `<wait>` followed by a `<throw>`.

The mapping of a sub-process depends on the message flows attached to it. If there is none, the sub-process will map to the `<invoke>` tag, which just calls a web service to fulfil the sub process. If the sub-process has an incoming message flow, it will map to a `<receive>` followed by an `<invoke>`. If it has an outgoing message flow, it will map to an `<invoke>` followed by a `<reply>`. If the sub-process has an incoming and outgoing message flow, a sequence of the `<receive>`, `<invoke>` and `<reply>` tags is used for mapping.

The mapping of the BPMN task element depends on the type of task, which can be set by attributes. A standard task maps to an `<invoke>` with the BPEL4WS input and output variable specified. A receive task will map to `<receive>`, whereas a send task will map to `<reply>` or `<invoke>` with just the input variable set. A loop task maps to `<while>`.

The data-based gateway maps to the `<switch>` tag, whereas an event-based gateway maps to the `<pick>` tag.

4.3.2 BPEL4WS Mapping Example

To show the mapping from BPMN to BPEL4WS a small part from the example section is chosen (see Figure 75). It is a part of the `Place Order Server Side` sub-process, containing activities, gateways and events.

Figure 75:   **Diagram example for mapping to BPEL4WS**
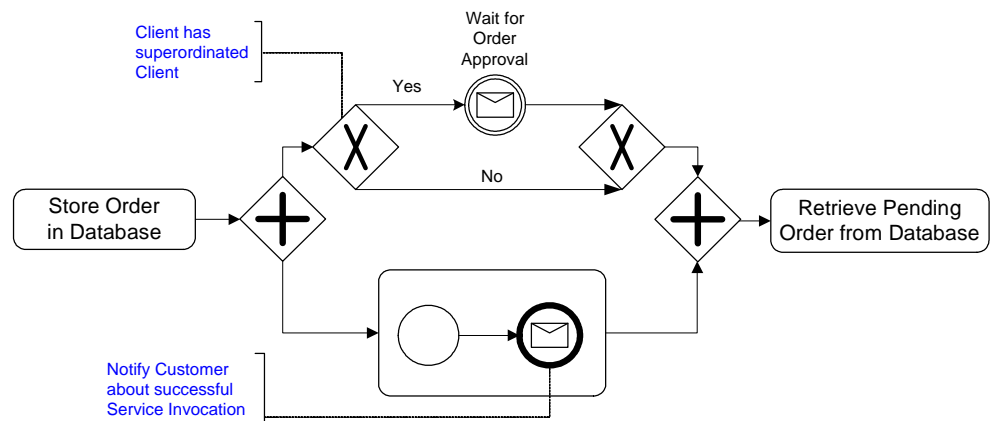
This example focuses on the mapping from BPMN to BPEL4WS. Therefore only the main flow is shown in Figure 76. Also, for clearness, some attributes are omitted. The BPEL4WS XML source code was derived from Figure 75 using the mapping rules shown in Figure 74.

```
<scope name="PlaceOrderServerSidePartly">

    <sequence>

   (2) <invoke name="StoreOrderInDatabase" inputVariable="processData"
               outputVariable="processData"/>

        <flow>

            <sequence>

       (4) (5) <reply
                name="NotifyCustomerAboutSuccessfulServiceInvocation"/>

            </sequence>

            <sequence>

                <switch name="ClientHasSuperordinatedClient?">

                    <case condition="Yes">

               (8) (9) <receive name="WaitForOrderApproval"/>

                    </case>

    (1) (3)            (6) (7)

                    <otherwise>

                  (10) (11) <empty/>

                    </otherwise>

                </switch>

            </sequence>

        </flow>

   (12) <invoke name="RetrievePendingOrderFromDatabase"
               inputVariable="processData"
               outputVariable="processData"/>

    </sequence>

</scope>
```
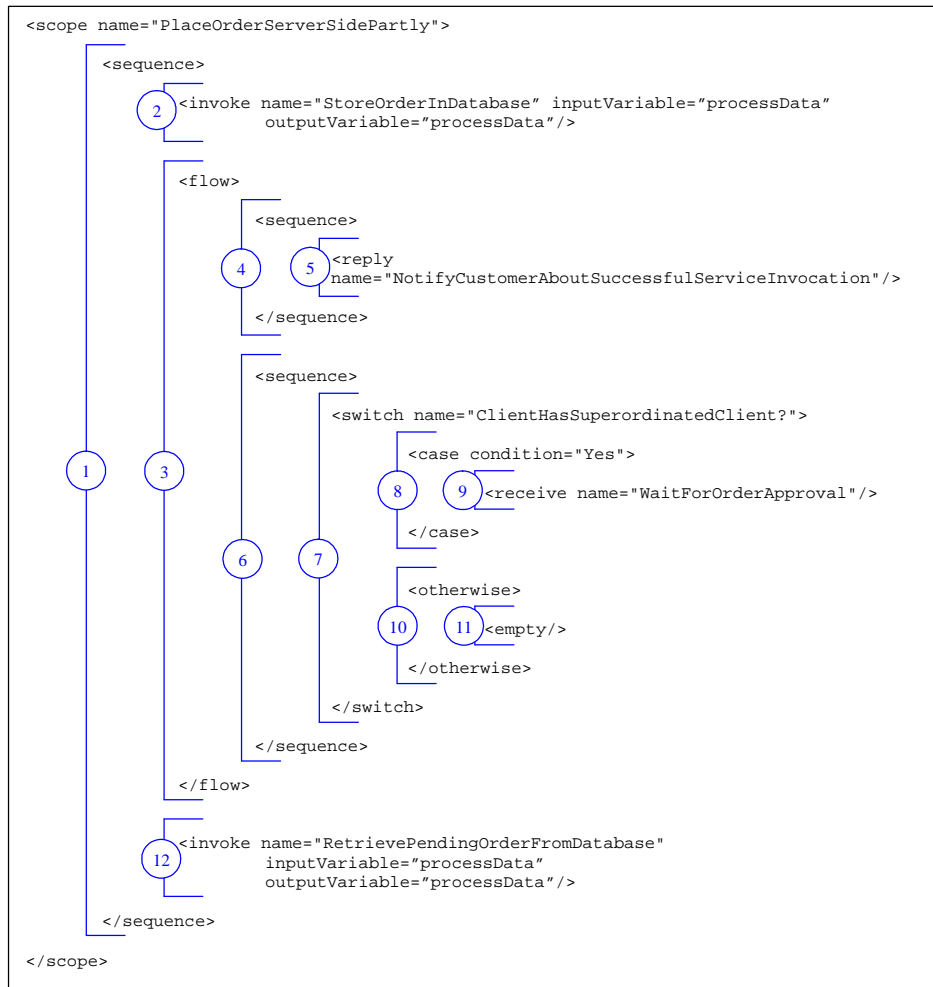
The main flow is surrounded by a `<sequence>` tag (1), because there is one starting and one ending activity, which indicate a serial flow. In (2), the `Store Order in Database` activity is called. A standard task has input and output variables set and is mapped to the `<invoke>` tag. Next follows a parallel fork, which is mapped to a `<flow>` tag (3). The parallel flow contains two serial flows, which are mapped to `<sequence>` tags (4, 6). The first flow (5) sends a message through a message end event, which is mapped to a `<reply>` tag. The second flow contains an or-gateway, which is mapped to the `<switch>` tag (7). The constraints are evaluated through a `<case>` tag (8). If the condition is true, a `<receive>` tag (9) is the fitting mapping for an intermediate message event in normal flow. Otherwise, nothing happens, which is declared in (10) by an `<empty>` tag (11). At last, the

76

`Retrieve Pending Order from Database` activity is mapped to another `<invoke>` tag.

4.4   Conclusion

The Business Process Management Initiative tries to establish a standard notation for business processes, which business users can easily understand. Also the notation can be mapped to execution languages like BPEL4WS.

The ease of use is reached through a basic set of core elements, which can be extended graphically and further configured by attributes. The BPMN draft specification contains an informal chapter of how elements can be mapped to BPEL4WS. The tools for mapping still don't exist; however there is a solution from Popkin Software (http://www.popkin.com) that supports modeling in BMPN based on a former draft specification.

BPMN is still under development, the draft specification itself names some problems that still have to be solved. This includes things like the behavior of transactions, a more formal mechanism of defining graphical extensions, new attributes as well as mapping to languages for abstract and choreography business processes (e.g. ebXML BPSS). BPMN still needs to be specified as a XML language layer above business process execution languages.

Technically, BMPN supports sequence and message flows; therewith it makes a clear cut between intra and inter-process communication. It can be used to model private, abstract and collaboration processes with different levels of detail.

The notation stays on a higher level without using things like object flow. Still, it is yet a draft and very little documentation beside are available. The strong needs for a standard notation for business processes exists and the three-part notation (core elements, extensions, attributes) makes a good start. However there is no mathematical foundation behind BPMN and the mapping to executable languages is still in the beginning.

5   Conclusions

This report presented three different approaches for modeling business processes. Generally speaking any of them can be considered to be of great expressiveness. However, BPMN is rather used for business modeling, while UML activity diagrams are more targeted at technical processes and software immediately implemented in software, while Petri nets are used for a wide range of purposes. The notation of any of these modeling techniques is specified clearly. Concerning the semantics of the elements only Petri nets dispose of a precise mathematically defined semantics, allowing for manual as well as computer aided semantics examinations. Based on their audience, which can be rather academic, technical or businesslike, the notations focus on different aspects of a process.

BPMN covers a wide range of high level abstractions like collaborations between different processes up to generic activities, whereas UML activity diagrams also cover technical details like object flow and storage. None of the presented notations provides explicit elements for the representation of variabilities of a process. UML activity diagrams and Petri nets support some kind of inheritance. However, more research has to be done in order to conclude this topic. The more recent languages UML activity diagrams and BPMN provide special elements for modeling concepts like events, interrupts, and exceptions. BPMN also provides mechanisms for specifying transactions.

To summarize, the UML covers a broad spectrum and is well established. UML activity diagrams provide a wide variety of elements making the activity diagram modeling more comfortable but also harder to learn. Also Petri nets are very well established featuring many tools, extensions, and literature. In contrast to activity diagrams Petri nets are easy to learn, but due to their limited means of expression less comfortable to handle. Both, activity diagrams and Petri nets get quickly cumbersome being used for modeling complex processes. BPMN is a newcomer, combining ease of use with a notation that is intuitively to learn. However, the applicability of BPMN in real world applications has yet to be shown.

References

[1]   Balzert, H.: *Lehrbuch der Software-Technik I: Software Ent-wicklung.* Textbook, Heidelberg: Spektrum-Verlag 2000

[2]   BEA Systems et al.: *Business Process Execution Language for Web Services Version 1.1*, May 2003

[3]   Berkenkötter K.: *Using UML 2.0 in Real-Time Development: A Critical Review.* University of Bremen, 2003. Internet-URL: *http://www-verimag.imag.fr/EVENTS/2003/SVERTS/PAPERS-WEB/04-Berkenkoetter-UMLRT-critic.pdf* [accessed in December 2003]

[4]   Björkander, M., Kobryn, C.: *Architecting Systems with UML 2.0.* IEEE Software pp 57-61, July 2003. Internet-URL: *http://www.uml-forum.com/out/pubs/IEEE_SW_Jul03_p57.pdf* [accessed in December 2003]

[5]   Born, M., Holz, E., Kath, O.: *Softwareentwicklung mit UML 2.* München: Addison-Wesley 2004

[6]   Business Process Management Initiative (BPMI): *Business Process Modeling Notation Working Draft 1.0,* August 2003

[7]   Chonoles, M. J., Schardt , J. A.: *UML 2 for Dummies.* Indian-apolis: Wiley 2003

[8]   Eshuis H.: *Semantic and Verification of UML Activity Dia-grams for Workflow Modelling.* Ph.D.-thesis No. 02-44, Centre for Telematics and Information Technology, University of Twente, Enschede, 2002

[9]   Kramler, G.: *Overview of UML 2.0 Abstract Syntax.* Techni-sche Universität Wien. 2003. Internet-URL: *http://www.big.tuwien.ac.at/staff/kramler/uml/uml2-superstructure-overview.html* [accessed in December 2003]

[10]   OMG: *Unified Modeling Language: superstructure. Version 2.0.* 2003. Internet-URL: *http://www.omg.org/cgi-bin/doc?ptc/2003-08-02* [last accessed in October 2003]

[11]   Petri, C. A.: *Kommunikation mit Automaten.* Institut für Instrumentelle Mathematik, Schriften des IIM Nr.2, Bonn, 1962

[12]   Schnieders, A.: *Application of web service technologies on a b2b communication platform by means of a pattern and UML based software development process.* Diplomarbeit, Technische Universität Berlin, 2003

[13]   Trompedeller, M.: *A Classification of Petri Nets.* 1995. Internet-URL: *http://www.daimi.au.dk/PetriNets/classification/* [accessed in December 2003]

[14]   van der Aalst, W.M.P., van Hee, K. M.: *Workflow Management: Models, Methods, and Systems.* Cambridge: MIT Press 2002

[15]   van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., et al.: *Workflow Patterns.* Department of Technology Management, Eindhoven University of Technology

[16]   W3C: *Web Services Description Language (WSDL) Version 1.1*, March 2001

[17]   W3C: *XML Path Language (XPath) Version 1.0*, November 1999