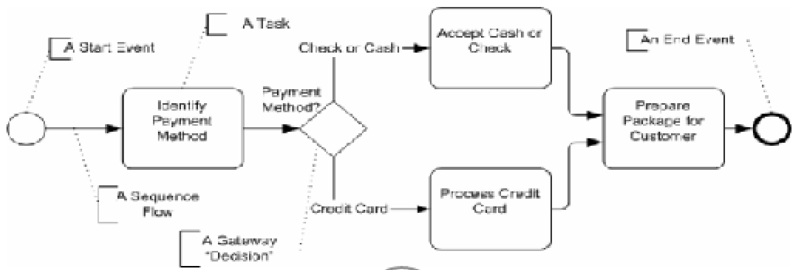
λx.



itasks
dynamic workflow system

Strongly typed BPMN

A functional, formal implementation of BPMN

Master thesis Information Science number 133 IK
Radboud University Nijmegen, The Netherlands
Department of Computer Science, section Model Based System Development

Written by Freek van den Berg (s0517593)
Supervised by Rinus Plasmeijer (aided by Bas Lijnse and Erik Crombag)

Spring semester 2010

Abstract:
BPMN stands for Business Process Modeling Notation and deals with offering a language in which business processes can be modeled. This thesis tries to answer the question "How can the BPMN meta model be expressed in terms of a functional language?" The first step that was taken is to formalize the BPMN diagrams using the ORM language. This includes everything to support a reconstruction of a BPMN diagram graphically. In order to support an implementation another addition to the meta model was needed and made, dealing with the data layer, the kind of implementation per atomic activity and some other concepts such as triggers. The Itasks library for Clean has then been used to implement BPMN examples by defining a mapping from the extended BPMN meta model to executable Clean code. This implementation has been validated by comparing the scenarios the software produced with manually generated use cases. This thesis contributed scientifically by making it possible to reason about BPMN constructs in advanced ways and by giving feedback to the Itasks community. Social relevance is achieved by making it possible to translate BPMN diagrams to working implementations and inspiring software vendors to use concepts as addressed in this thesis.

Strongly typed BPMN

A functional, formal implementation of BPMN

Table of Contents

1 Glossary

An overview of terms and concepts commonly used in this document and their corresponding meaning is as follows:

Term	**Description**
Activity	A piece of work in the BPMN notation, possibly having input and output. Subtype of a flow object.
Atomic activity	An undividable piece of work in the BPMN notation.
BPEL	Business process execution language: A systematic language normally used as intermediate language to connect a business modeling language with an implementation language.
BPMN	Business Process Modeling Notation: An intuitive, commonly used modeling language in diagram form to describe business processes.
Business process	A set of generalized activities that are connected in a certain way and are supposed to solve a business (sub)problem or reach a business goal.
Clean	A functional programming language that supports lazy evaluation, higher order functions and strongly typing.
Compound activity	A composition of activities, which is itself an activity again in the BPMN notation.
Connecting object	A BPMN object to connect different flow objects to each other.
Event	Something happening outside the business process in the BPMN notation, that causes an effect on the process.
Flow object	A BPMN object in which something happening in the “real” world is represented.
Google maps	A web mapping service offered by Google. Includes an API to deal with map services in customer made applications and web sites.
Itasks	A library for the programming language Clean, which aids in creating a workflow system in a functional way.
JSON	Javascript Object Notation: A lightweight notation to describe objects in a similar way XML does.
Lane	A subset of a BPMN process which is implemented by a single system.
Lazy Evaluation	The calculation of a result is delayed as long as possible, until it is really sure that the result will be needed.
ORM	Object role modeling: A formal modeling language to conceptually describe objects and the relationships (fact types) they are involved in.
Polymorphism	A programming language feature that allows values of different types to be handled using an uniform interface. This prevents the need for having to describe functions for different data-types, but has the downside that what one can do with the elements is more restricted.
Pool	A subset of a BPMN process in which only certain resources are available.
RPC	Remote procedure call: Calling a function outside the current system in order to let a computation take place there. The result of the computation is returned.
Task	In Itasks: A piece of work that needs to be done, having an output. Very similar to the concept of activity in BPMN.
Trigger	An event occurring in the outside world that impacts the system and requires something to happen internally or vice versa.
Workflow	In Itasks: A decorated task that can be plugged into the engine of Itasks. Very similar to a business process in BPMN except for the part that a workflow is generally considered to be more dynamic and changeable.

2 Introduction

The introduction consists of three paragraphs, namely:

- Problem statement: What is the actual problem and what research question is the result?
- Method: The steps that are taken to solve the problem.
- Justification: The reasons why solving the problem is useful.

2.1 Problem statement: How can the BPMN meta model be expressed in terms of a functional language?

When is system or company is designed, it is generally useful when it can produce a certain output given a certain input. In a company this is normally translated to orders being the input, and delivering the actual orders as output. Because one normally wants to regulate what is happening in the system or company, a formal abstract world is generally introduced that models the relevant aspects that are going on in the real world. Normally the process between input and output is relatively complex and by naming intermediate checkpoints it becomes possible to see more accurately how far work has been progressed. Workflow systems tackle this problem by decomposing a bigger task that needs to be done (such as handling and delivering an order) in several sub tasks, which can run independent of each other in a sequential and/or parallel way. In many workflow systems the output of one task is connected to the input of a next task that needs to happen later in time.

The workflow systems that are on the market and that are named in scientific theories, generally do not deal with having the input of one task match the output of the previous task (strongly typing). Also the execution of workflows that are designed in a workflow system is very problematic or not possible. A commonly used standard for Business Process Modeling is the BPMN language, which offers a notation for business processes in diagram form.

The intention of this thesis is to combine the advantages that BPMN has (such as global acceptance, readability by a broad public) with the advantages the scientific theories offer (such as strongly typing and functional reasoning). This results into the following research question:

- How can the BPMN meta model be expressed in terms of a functional language?

In order to do so, the following sub questions will be considered:

- What current literature is there about workflows, BPMN and functional languages which is relevant for the main research question and what does it state?
- How does the meta model of BPMN look like in a formal way?
- What extra additions need to be made to the BPMN model in order to make sure its graphical representation can be reconstructed again?
- What extra additions need to be done to the meta model in order to support the data flow?
- What options are there for the implementation of an atomic task and how can they be added to the meta model?
- How can the extended BPMN meta model be translated to a functional language?
- What does this translation look like on some common BPMN examples?
- How can one have faith that this translation does what one wants it to do and how is it tested?

The next paragraph in which the method of this research is included, will show in what way an answer to the above questions is obtained.

2.2 Method: From the literature study, formalizatoon of BPMN to a validated Itasks implementation

The first step that is taken is a literature study in which all the relevant topics are comprehensively researched and briefly explained. Also a first attempt in connecting the different topics is made here. This leads to an overview of the scope of the domain this thesis is going to attend in.

The second step encompasses the formalization of the BPMN notation. This step involves looking at sample BPMN diagrams, reading the specification and translating the concepts as presented to a correct ORM model. This results into a BPMN meta model. Some design decisions need to be taken here, since the BPMN notation is not completely formal and unambiguous on this aspect and attempts have been taken in order to do so already[19][20]. As an extra step in addition to just mentioned, some data about the positioning of the objects and their size might need to be stored in order to graphically reconstruct the original BPMN model given the meta model. This can be done by extending the meta model with that extra information, but is not done in this thesis, because this information is not relevant for the implementation.

Third an underlying data model is created to support the execution of BPMN. The BPMN diagrams on itself do not reveal how the data flows are going through the process and merely show a sequence in which activities have to be performed. A solution could be to transfer all the data along every step of the process, but it is nicer to have every step of the process only have access to the data it needs. The exact examples of this approach will be discussed here as well and the result of this leads to an extension of the BPMN meta model

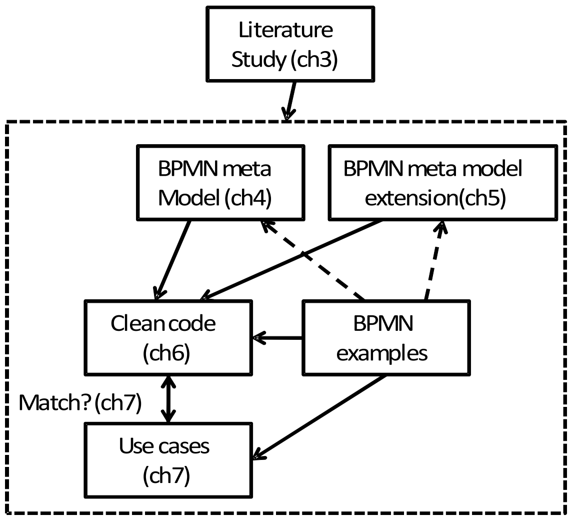
Fourth different alternatives are investigated about how to support an atomic task, which in the functional setting turns out to be a transformation from a set of certain input types to a set of certain output types. It is important here to support different ways and to be as compatible as possible with both the functional element and the ability to create different atomic tasks at different systems. This is also part of the BPMN meta model extension.

Fifth the implementation is discussed. The extended BPMN meta model (so step 2,3 and 4 taken together) is mapped to a functional language by providing an algorithm how to do so. Example BPMN diagrams are used, resulting in concrete code in the functional language.

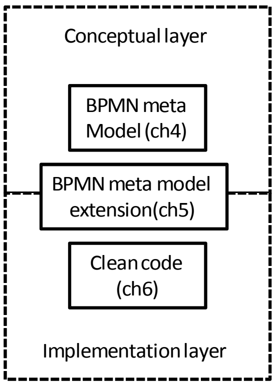
In order to have confidence that what happens in step five, the implementations are ran and compared with use case diagrams that are generated from the BPMN diagrams, which intuitively state what interactions take place. Ideally the implementation presents the behavior as described in the use cases.

Finally a chapter is dedicated to answering the main research question using all the answers of the sub questions. Also extra things that can be concluded from every step as just mentioned are added to the conclusion whenever found valuable. The problems encountered and the possibilities gained are discussed, as well as the relevance of this document. Through a paragraph about future work, challenges that remain and work that can be done in the future on this field are expressed.

In diagram form this leads to the following picture (with the corresponding chapter numbers included):



With respect to the conceptual and implementation aspect chapter 4,5 and 6 can be placed as follows:



The initial BPMN meta model is mainly conceptual, the extension partly implementation dependent and the Clean code deal with implementation aspects only.

2.3 *Relevance: Offering a formal BPMN model, feedback about Itasks, implementing BPMN*

The thesis will have both scientific and social relevance. The scientific relevance becomes clear when the BPMN meta model is generated and transformed to an executable solution in a functional language. This makes it possible to reason with the BPMN language in a formal way and to extend it with constructs that are available in a functional language and by using higher order functions. Also semantics can be given to BPMN constructs and effects of BPMN constructs at implementation level can be investigated.

In addition to this, this thesis might be found beneficial for the Itask community at the Radboud University, which also provides scientific relevance. The people of the computing science department and more specifically the model based system development research group[R18], have been working on a functional implementation of workflow systems and have tried to map sample business cases to their system. However due to the distance that exists between management people and computer science people this has not always been easy to communicate. By mapping BPMN to the Itask system, communication towards management people becomes possible and it becomes more clear where the Itask system is standing with respect to the current demands in workflow theory.

The social relevance is expressed when the graphical notation of BPMN, which is normally known or familiar by various kinds of people, can be used to show a manager what a work flow system is actually doing. When the BPMN meta model is linked to a GUI, it becomes possible for a non-IT person (which a domain expert often turns out to be) to design and change his business in an intuitive way with direct results at the implementation level.

Finally it might become possible to reduce the design and implementation costs of BPMN systems in general and for businesses to respond quicker to the market. The latter aspect is achieved by loose coupling the IT infrastructure and business layer by having an intermediate layer, which causes both of them not to depend on each other too much anymore.

2.4 *Conclusion*

This chapter has shown that the design of a system or company, normally requires a view in which input is transformed into output, using resources. In order to control better what is going on, normally intermediate transformation steps are defined, of which the output is connected to the input of the next step. This is called a workflow. BPMN offers a language to describe this in diagram form, but has the disadvantage of being informal. The scientific theories offer strongly typing and functional reasoning and the intention of this thesis is to combine BPMN with them, by answering the research question: How can the BPMN meta model be expressed in terms of a functional language?

A seven step method is conducted to achieve this, encompassing the making of a BPMN formalization, implementing it in the Clean language using the Itasks library and testing it using use cases. After this both scientific and social relevance of the thesis are discussed. Scientific relevance is achieved by the formalization of BPMN and by contributing feedback to the Itasks community with respect to the Clean implementation. Social relevance is achieved by the mapping between a BPMN diagram and its implementation, which makes it easier to show and create working software and by addressing constructs that could be implemented in real businesses.

3 Current Literature

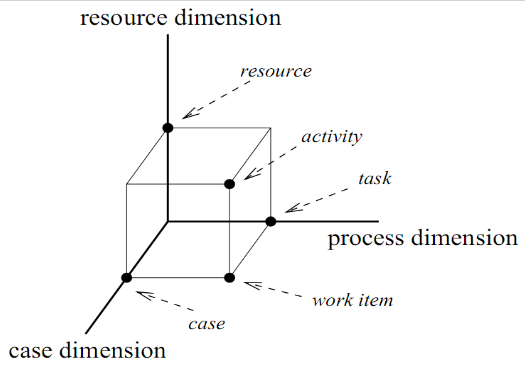
The literature study is done in five steps. First all the theory that is available concerning workflows in general is being discussed. This leads to an overview of what workflows really are. After this BPMN, a graphical language that deals with workflows is discussed. The functional characteristics workflows have are the third part of the literature study. Fourth the current practice of workflows is mentioned, including the tool CPF by Cordys and BPEL. This is done, in order to have some inspiration and a starting point in translating BPMN to executable programs. The literature study is concluded with a brief topic about communication, in order to make it possible to have different systems communicate and connect.

3.1 Literature about workflow theory

Firstly Ashby's system theory[1]. In his theory Ashby distinguishes three stages in managing a system named respectively control, design and regulation. The first stage control deals with identifying the essential variables and the norms they need to satisfy. An example is a system being a human being, a variable its body temperature and its corresponding norm 35-42 degrees Celcius in order to stay alive. Identifying the variables leads to an abstract system besides the real system, which state is defined by taking the values of all variables together. Normally this system is less complex then the original system one has to manage. The second stage named design encompasses the detection of treats to the essential variables named disturbances. Disturbances can cause the values of the variables to leave the norm interval and therefore require regulatory actions to prevent this. Sticking with the human example a disturbance might be the flu which can cause a high fever (42+ degrees Celcius body temperature) and regulatory actions are seeing a doctor, taking some rest or taking a medicine. Finally there is the regulation stage in which everything from the first two stages has to be applied in practice. This requires creating the right infrastructure for detecting disturbances and applying the right regulatory actions in a timely fashion. When Ashby's system theory is translated to the domain of workflows, control and design mostly deal with creating and maintaining workflows in order to see what needs to happen. On the other hand regulation is concerned with the actual execution of the workflows, which means the right infrastructure needs to be available to make sure that what is supposed to happen, will happen.

Secondly there is De Sitter's paper about organizational design[2]. With respect to dividing work into tasks there are two extremes that can be mentioned. One one hand there is Taylorism[21][22] (in the paper described as a complex organization with simple jobs) in which all the work that needs to be done is divided into tiny units which are very carefully described. This approach is also known as Scientific Management. The advantage of this approach is that it is very clear what needs to be done, intermediate results can be measured easily and employees or resources do not need to have special qualities, since creating the workflow and defining the tasks is where the thinking and room for creativity takes place. The downside of this approach is having bored employees because everything they do is very fixed and there is no room for creativity. In addition to this the approach makes a system very inflexible to changes, causes a lot of bureaucracy for having all the small tasks connected and makes it likely for errors to occur because of the large amount of communications between the small tasks. The opposite of Taylorism[23][24][25] (also known as an organic structure) is to have a simple organization with complex jobs. In this scenario tasks are not clearly defined and often assigned to a group of people with various qualities. This group of people is free to solve the work any way they want as long as the results meet certain standards and people within these groups normally have several core competencies. This approach leads to challenging work with a lot of variety, but has the disadvantage that it is hard to check what the current state of the system is and whether progress with the work has been made. Also employees might feel overwhelmed by the complexity in case they do not see what is going on. According to De Sitter, starting from the really big task that needs to be done, tasks can be split in three different ways. First of all one can split a task by separating the managing part and the execution part. This means that one half is responsible for the actual work being doing (executing) and the other half for thinking of ways how to do get it done and monitoring the performance (managing). Besides this option, it is possible to split using parallelization and segmentation. As an example to explain this a bicycle store is taken in which bicycles and mopeds are fixed. Imagine there are two people working in this store. Initially both people just take the first order from the list that comes in. This requires them to both know how to both fix bicycles and mopeds. When one person is specialized in bicycles and another in mopeds, it makes sense to apply parallelization to the work flow. This means there will be one task for fixing bicycles and one for mopeds. On the contrary segmentation occurs when fixing a vehicle (either bicycle or moped) is split into two sequential tasks. One person could for example diagnose the problem and do the preparation (eg. putting the tools in place, after which the second person finishes the job. De Sitter prefers to use parallelization more above segmentation, because in the case of segmentation when the two people talk, their communication becomes more complex (eg. Is the other person talking about bicycles or mopeds right now).When De Sitter's theory is taken into account regarding designing and implementing workflows, it becomes important to consider how complex the workflow should be, where separation in smaller tasks should take place and who is responsible for designing and executing which part of the work flow.

Thirdly Van der Aalst's papers[3][13][27] in which he makes a distinction for workflows in three dimensions which are the resource, process and case dimension (see the diagram below). The process dimension deals with the way work is done (a task). This is normally what workflows address. When a specific case of this work (for example an order for a product) is considered, the case dimension is entered. A work item is created when the process and case dimension are combined. This means that when all work items of a system are taken together, one can see all the work that needs to be done. This knowledge is however not sufficient for managing the system well. Generally the work that needs to be done requires resources (for example a human or machine) that can only do one or a limited number of tasks at the same time. This requires scheduling since it is not always possible to perform all the work that needs to be done right away. When the resource dimension is added to a work item and thus then all three dimension are represented, one talks about activity. The three dimensions are important with respect to workflows, because they make it possible to think about how everything is scheduled in addition to the traditional approach which is mostly concerned with how the work is done:



Fourthly Proper's lecture notes named Ground Enterprise Modelling[51], which are part of the DaVinci Series. A literal description is given for a workflow, namely: The flow or work from in the system. This matches the view that work flows through a system and that it is a path between the input and the output of the system. In addition to this the terms work process and work case are mentioned. A work process is stated to be 'a flow of work taking place in a work system' while a work case is mentioned to be ' A work process taking place in the work-system under consideration, which is triggered by an external trigger'. These definitions match the just mentioned Van der Aalst's definitions very much and are applicable to BPMN later on.

3.2 Literature about business process modeling notation

Looking at workflows functionally is compatible with the approach the Business Process Modeling Notation language (from now on referred to as BPMN), a commonly used standard for describing business processes, is taking. In the paper by Terje Wahl[6] BPMN is evaluated and also others have done this[28][29]. The conclusion is that the language is very much functionally oriented, applicable to the business domain as well as other domains and that the diagrams that can be produced with the language are easy to understand. Especially the final aspect is very important when one wants business people both to understand the workflows and actively think about what is needed. Because of this reason it is important to consider the BPMN whenever thinking of workflows or business processes.

The BPMN notation is explained from the scratch in Stephen White's paper[7]. A distinction is made in four kinds of objects, namely flow objects, connecting objects, swim lanes and artifacts. The flow objects generally contain the work that needs to be done. They are connected to each other (to indicate a sequence) using connecting objects. Swim lanes offer support for flows in which more persons or other work performing entities (resources) are collaborating. Finally artifacts are extra objects to make more clear what is going on in the flow objects field, such as data that is being transferred between them or informal annotations. What the BPMN notation is lacking however, is a formal definition and clearly defined semantics. As a result of this the execution of what is made in the BPMN notation, is not straightforward and certainly not yet automatable.

Weaknesses the BPMN notation has are its ambiguity (which this thesis will counteract by introducing formal definitions) and the ability to convert the models to executable environments (which is shown to be possible using Itasks). Alternatives for BPMN which have been considered but found less suitable are UML[47][R29], BEMN[48], Petri nets[49][R30] and CEP[50].

3.3 *Literature about functional thinking and languages*

My own bachelor thesis offers a taxonomy and meta model for describing production systems[4][R9]. The meta model is made in the formal ORM[30][31][32] language and contains an object type named 'production step description'. This entity is comparable to what Van der Aalst calls a task and is mostly concerned with the process dimension. Connected to this entity is an input and output of so called 'product requirements'. Since this part of the meta model deals with the abstract world, the input and output of this entity are not actual physical products. The connection is mainly just saying that when input products meeting certain requirements are given, the production step will guarantee an output product meeting requirements. This concept is very similar to the signature of a mathematical function in which numbers of a certain input (real numbers, whole numbers, positive whole numbers etc.) guarantee that the function will give an output of a certain kind. It might therefore be useful to use the functional approach for creating workflows.

This approach is taken in the Itasks system[5], which is a scientific framework and programming library for designing and executing workflows web based. The I-tasks system is developed in the functional language Clean and designing workflows is achieved by writing functions in the language Clean. Other functional languages, which have been considered to be used in this thesis, are Miranda[37], Camel[35][36], LISP[54][55][R37] and Haskell[33][34][R19], which like Clean all find their foundations in the lambda calculus[56]. However, they do not offer a system with similar functionality as Itasks system. Some main properties about the code that can be written and its execution are the code being strongly typed, lazy evaluated, higher order and functional (so without side effects). In addition to this multi users (or resources) and constructs that are used in many programming languages (such as recursion) are supported. This makes the Itasks system very suitable for formally reasoning about and for testing workflows.

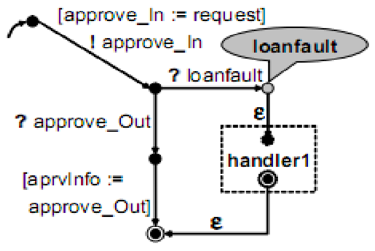
3.4 *Literature about executable business process solutions*

For inspiration purposes the Cordys Process Factory [8][R1] has been used, which is a completely web based application in which one can design and execute applications and which is developed by Cordys[R4]. The application has been built using a model driven approach, which means getting the model straight is the first priority and what needs to be executed from it often occurs naturally and is relatively simple. A similar approach will be taken in this thesis. The designing of applications is partly done by drawing BPMN graphs and assigning web services to the building bricks of BPMN. The result is an executable application without doing any extra effort on the implementation level. The web services can either be entities anywhere in the world or locally defined web services that access the local data structure of the application's database. This approach makes it possible to make virtually any business process one can imagine and to monitor its different instances. The system is however lacking type checking, since every output can be mapped to every input without a check for whether data types match being performed (since every data type is a string and thus always matches). This can lead to errors later on when the system is already up in the air. In addition to this, this approach lacks an easy 'User interaction' task. The proposed user prompt task feeds data to a user of the system and asks the user to transform the data in a certain way (Eg. On the question “How old are you?” the user might reply “26”). The benefit of this task above a web service, is that one can quickly build them and does not have to have access external entities outside the system to use them. Appendix A provides a brief impression of what the Cordys product is like by providing screenshots with a small description.

Also the BPEL [10] language has been considered and kept in mind, but generally been found unsuitable because of its lack of readability. Diagrams in the BPEL language generally look like programming language (see diagram below) that is executed on a computer and put a high learning curve on what the business manager is supposed to know. They are easy to translate to petri nets however and can certainly be used t as an intermediate language between BPMN and an implementation language[16][17][18]. Due to the functional approach that has been chosen in this paper however, a direct translation to a functional language is preferred.

invoke

```
<scope>
 <invoke ...
  operation="approve"
  invar="request"
  outvar="aprvInfo />
 <catch ...
  faultname="loanfault"
  < ... handler1 ... />
 </catch>
</scope>
```



3.5 Literature about communication between systems

When the business system becomes part of a bigger whole, communication with other systems becomes important. At the time of writing XML[38][39] is one of the standards to format messages, which is used in webservices that encompass WSDL[40][41] (the definition language) and SOAP[42] (the messaging framework). In Clean however, a more light weighted RPC[43][44] Daemon is used which expects messages back in JSON[45][R20], which stands for Javascript Object Notation. This notation can be parsed and type checked by the Clean environment and can also be sent to the user without exactly knowing how the interface deals with the data type. The latter option will therefore be used in this thesis, but in order to remain compatible with most systems, support of the former one is highly recommended. A comparison between XML and JSON[R31] looks as follows:

Language	**Example**

JSON	`{`
	`  "glossary": {`
	`    "title": "example glossary",`
	`    "GlossDiv": {`
	`      "title": "S",`
	`      "GlossList": {`
	`        "GlossEntry": {`
	`          "ID": "SGML",`
	`          "SortAs": "SGML",`
	`          "GlossTerm": "Standard Generalized Markup Language",`
	`          "Acronym": "SGML",`
	`          "Abbrev": "ISO 8879:1986",`
	`          "GlossDef": {`
	`            "para": "A meta-markup ... languages such as DocBook.",`
	`            "GlossSeeAlso": ["GML", "XML"]`
	`          },`
	`          "GlossSee": "markup"`
	`}}}}}`

XML	`<!DOCTYPE glossary PUBLIC "-//OASIS//DTD DocBook V3.1//EN">`
	`  <glossary><title>example glossary</title>`
	`    <GlossDiv><title>S</title>`
	`      <GlossList>`
	`        <GlossEntry ID="SGML" SortAs="SGML">`
	`          <GlossTerm>Standard Generalized Markup Language</GlossTerm>`
	`          <Acronym>SGML</Acronym>`
	`          <Abbrev>ISO 8879:1986</Abbrev>`
	`          <GlossDef>`
	`            <para>A meta-markup ... languages such as DocBook.</para>`
	`            <GlossSeeAlso OtherTerm="GML">`
	`            <GlossSeeAlso OtherTerm="XML">`
	`          </GlossDef>`
	`          <GlossSee OtherTerm="markup">`
	`        </GlossEntry>`
	`      </GlossList>`
	`    </GlossDiv>`
	`  </glossary>`

What can be seen from the above examples is that both languages express the same concept, but that JSON saves a little space but omitting closing tags. In addition to this the WSDL/SOAP solution for webservices provide a lot of overhead by requiring a lot of meta information.

3.6 Conclusion

The literature study has made clear what different views there are on workflows in the literature, which is used implicitly all the time during the thesis when design decisions need to be made. As a business modeling language BPMN is very suitable due to its easy to understand and the wide support it has globally. There are alternative languages such as UML, BEMN and petri nets, but they will not be considered in this thesis. When a functional approach is wanted, Clean turns out to be a very good language to model this. Very much because of the availability of the Itasks system, which is a functional system especially designed for workflows. With respect to executable solutions, The Cordys Process factory offers good examples which can be used for inspiration. For communication an RPC call in JSON format will be used, mostly because is easy and lightweightness, but in order to adopt to standards a support for WSDL/SOAP webservice is highly recommended.

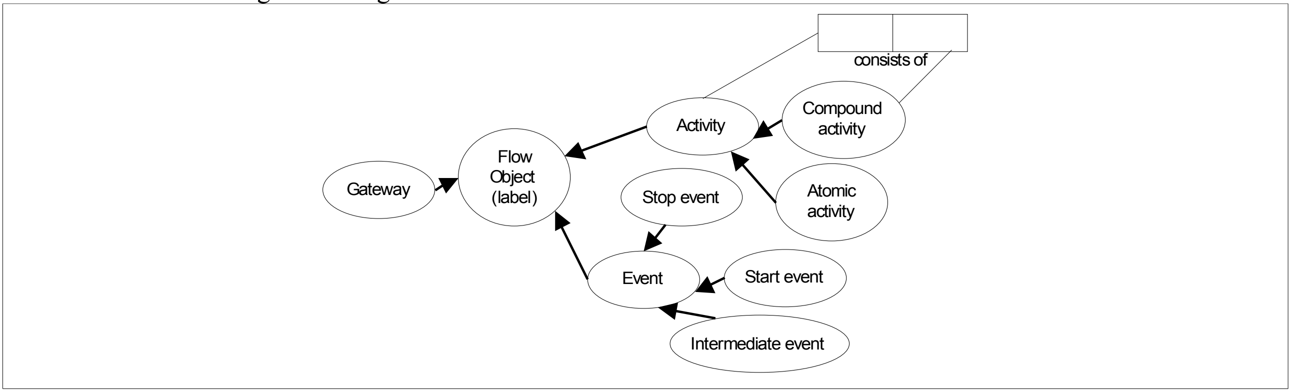
4 Creating the BPMN meta model

As mentioned before in the literature study the BPMN notation is a very commonly used notation which is easy to understand by a big group of people ("The primary goal of the BPMN effort was to provide a notation that is readily understandable by all business users"[7] and "It is a goal for BPMN that it should be understandable not only by IT professionals, but also for business analysts and other non-technical people"[6]). This chapter will introduce the BPMN language more extensively and will try gasp the formal meaning of the language by making a meta model in the ORM language. The creation of the meta model happens iteratively, after which the different smaller steps are combined to create one single bigger meta model. Describing a BPMN population in this language makes it possible to describe transformations to other languages (for example by using XML as an intermediate language). However, in this thesis the transformation step is done informally, but in a very structured and automated way. This model driven approach might initially cost some more effort then other approaches, but will in the long run make the gap between diagrams and their implementation smaller.

As a reference the earlier mentioned BPMN notation paper[7] is used. Different kind of objects are described in this paper and grouped in a certain way. First the focus will be on the grouping of the objects, after which through examples their use is illustrated and the way they connect determined. In the paper the BPMN language consist of four groups of constructs, which each have their own objects. The group of constructs are flow objects, connecting objects, swim lanes and artifacts, which will be illustrated by examples later. The corresponding objects for every group can be expressed in ORM by using subtyping.

4.1 *Adding flow objects to the BPMN meta model*

For flow objects, which are considered the objects in which the real actions occur, and having the specialization event (either start, stop or intermediate), activity (either atomic or compound) and gateway belonging to this group, this results into the following ORM diagram:

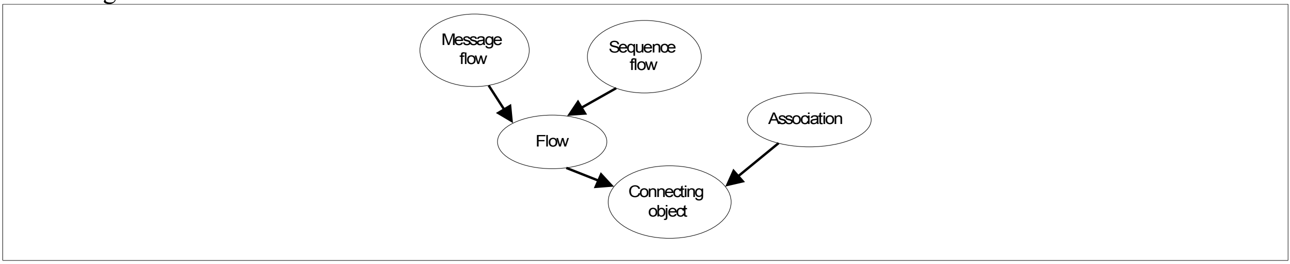


Note that a fact type has been added for composition. The fact type ensures that a compound activity can be made, just by stating which activities it encompasses. In a one level composition these would be atomic activities only, but it is imaginable that compound activities once again contain compound activities and so forth. Recursive definitions are not desired here though. The label entity has been added at the highest level of the subtype hierarchy (flow object) for two reasons:

- First it is required to uniquely identify every flow object.
- Second as every flow object ,which will be illustrated later the BPMN notation, has a piece of text besides it, it will need to be stored.

4.2 *Adding connecting objects to the BPMN meta model*

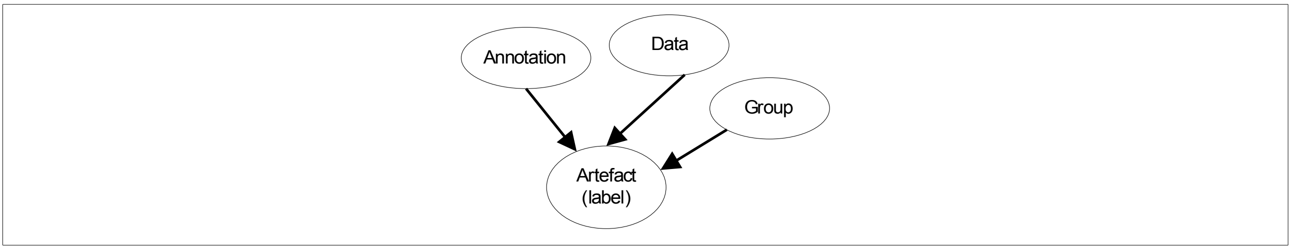
For a connecting object consisting of flows (either sequence or message) and associations this means the following ORM diagram:



The connecting objects serve two purposes, namely the connection of flow objects to generate a sequence and associating an artifact to flow objects in order to provide some extra information about what is going on in the diagram.

4.3 Adding artifacts to the BPMN meta model

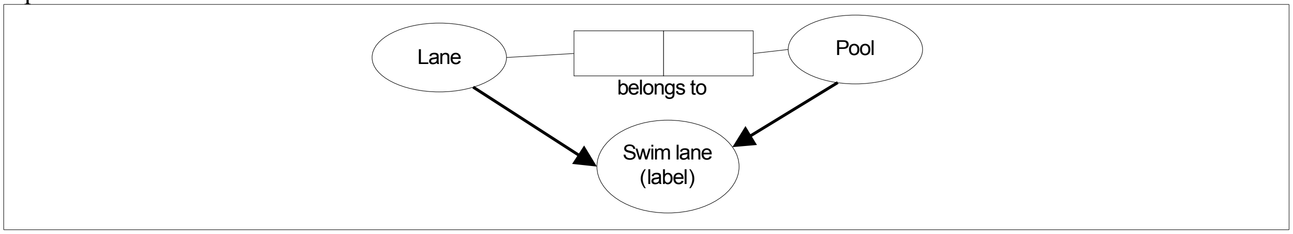
For artifacts the result is:



Artifacts not need to to have a formal meaning in the implementation (although data and group could in some way), but are mainly meant to illustrate more clearly what is going on in the diagram. This happens in a similar way as comments or annotations clarify code in a programming language.

4.4 Adding swim lanes to the BPMN meta model

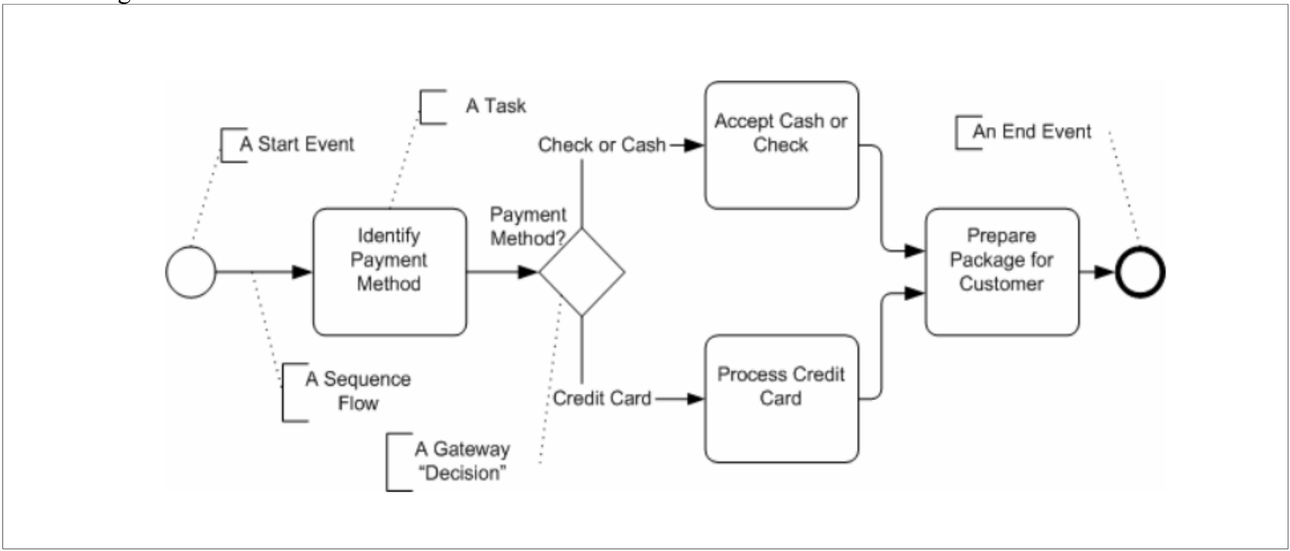
Finally the swim lanes. They consist of lanes and pools and a lane is a partition of a pool. In ORM this can be represented as follows:



Metaphorically the differences between lanes and pools can be described as follows: One swimming pool consists of different lanes in which people can swim. The lanes however do share the same water in contrast to different pools. The water can be seen as the resources that are shared and as turns out later in this thesis. For the implementation having different lanes means having different entities working on the same system, whereas different pools symbolize a completely different system at implementation level.

4.5 Defining the roles of connecting objects in the BPMN meta model

Now that all objects have been mentioned, it is possible to figure out how they are related to each other. This is done by using a few examples in which it becomes clear what the different objects represent and how they are connected to each other. In other words: The roles the objects play with each other, also known as fact types in ORM. Also the examples make it possible to address a population to the fact types and object types, which makes the whole idea a bit more lively. This is especially important later on when the implementation is taken into account. Consider the following BPMN diagram:



The population that can be addressed in terms of everything that has been shown in ORM so far is:

```
Start event = {''}

Stop event = {''}

Atomic Activity = {'Identify payment method', 'Accept cash or check', 'Process credit
card', … }

Annotation = {'A start event', 'A task','A gateway decision', 'An end event'}
Gateway = {'Payment method?'}
```

The 'check and cash' label and the 'credit card' label have not been mentioned yet. In order to simplify the BPMN notation, the 'Payment method?' is changed into 'Payment method credit card?' after which through yes and no paths a Boolean question remains. This will keep the data flow and activity flow as loosely coupled as possible. The '' stands for an instance without a name, as in the above diagram the start and stop event (also known as end event) are not labeled.

What is left is to describe the different connections between the entities. As its name indicates, a connection object connects things. It is worth making a distinction between a flow, which connects two flow objects, and an association, which connects an artifact and a flow object. Also the just mentioned gateway needs extra attention, since it has outputs that need to be distinguishable. This is done by giving it an extra role, which stands for the true path in addition to the false case (which is considered to be default). The result of the gateway can be determined by the task preceding it and seeing in which path the result appears. In this case the preceding task is the 'Identify payment method' task.

It does not necessarily need to be the case that the preceding task determines the result of the gateway. Some extremes are (which of course can be combined):

– A coin is flipped somewhere at random, which decides which path of the gateway is taken
– A huge calculation is done which depends on all the work that has been done so far. The result of this calculation determines which path of the gateway is taken
– The same path of the gateway is always taken no matter what. This means the other path of the gateway is just there for cosmetic (and probably confusing) reasons

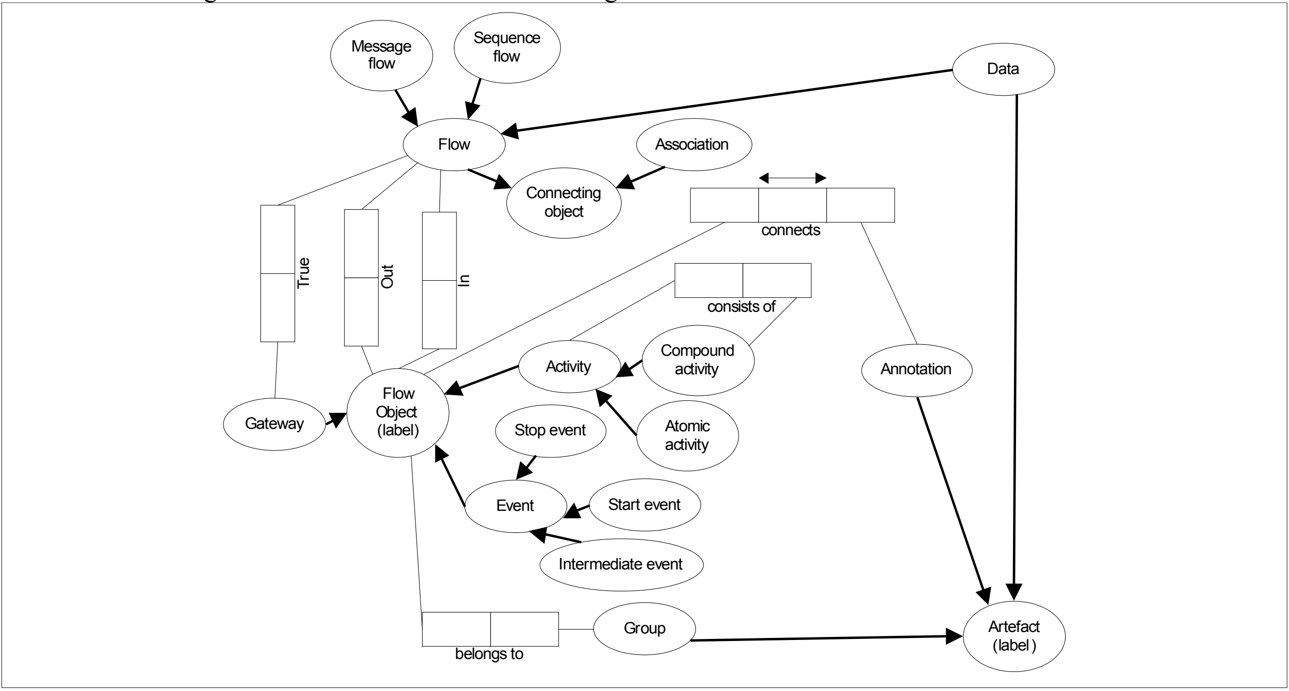
However, what will turn out later in the data layer model, is the decision to make the gateway deal with the task preceding it per se. This in order to make the gateway function part of the meta model easily, namely as a filter on the data type preceding it.

Finally, in order to simplify the model, the 'A sequence flow' artifact is ignored, because it is pointing to a flow instead of a flow object. This seems sensible to do since connecting objects are merely meant to connect different flow objects in which the things really happen. For the implementation it does not matter anyways, since artifacts have no effect on the functioning of the program.

In short the fact types (read: roles) that need to be added are:

– 3 different connections between flow and flow object (in, out and true for a gateway)
– A connection between a flow object and an artifact to support an association

The result of adding these connections to what has been given before in ORM is:



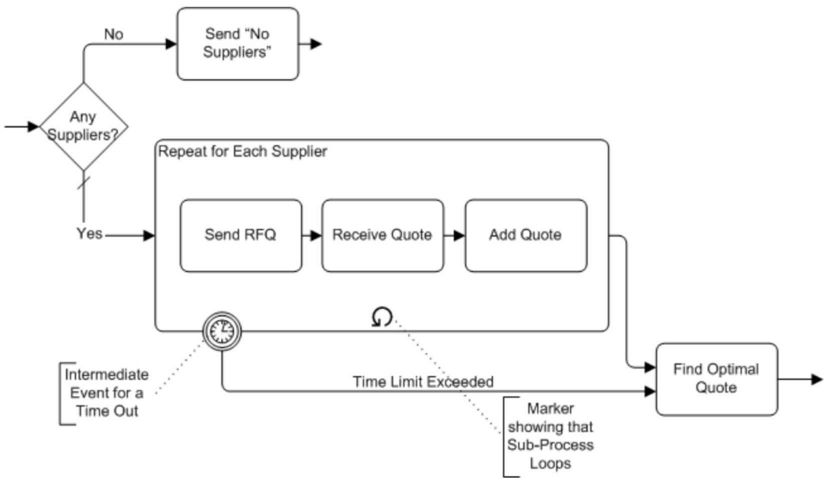
The population from the rules given the diagram noted at the beginning of the paragraph then becomes (in addition to what has been given before):

```
Flowobject-Association-Annotation artefact = {('','1','A start event'),('Identify
payment method','2','A task'),('Payment method?','3','A gateway decision'),('','4','An
end event')}

Flow-Flowobject(In) = {('Identify payment method','1'),('Payment method?','2'),
('Accept cash or check','3'),('Process credit card','4'),...}

Flow-Flowobject(Out) = {('','1'),('Identify payment method','2'),('Payment
method'.'3'),...}

Flow-Gateway = {('Payment method','4')}
```

In case the original BPMN diagram does not show a name for an entity (eg an association is just a line), integer values have been used to make them identifiable. Any other way of identification (such as the both items its connecting) would have been sufficient as well.

4.6 Defining the roles of compound activities in the BPMN meta model

The next diagram uses some more advanced concepts of BPMN and can once again be translated to facts (resulting in fact types at meta level) describing them:

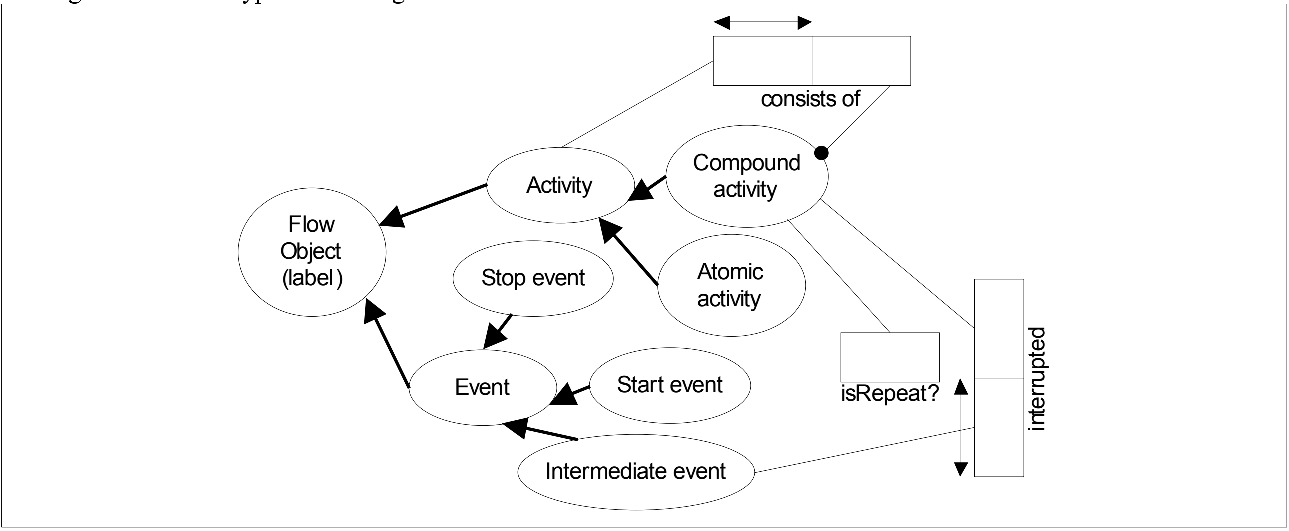


Note that the compound activity encompasses three smaller activities in the above diagram and that it is a special activity that "loops".

This results in the following fact types:

- A compound activity contains one or more flow objects
- A compound activity can have an intermediate event as output
- A compound activity can be looped over a data set (by adding the marker)

Adding the three fact types to ORM gives:



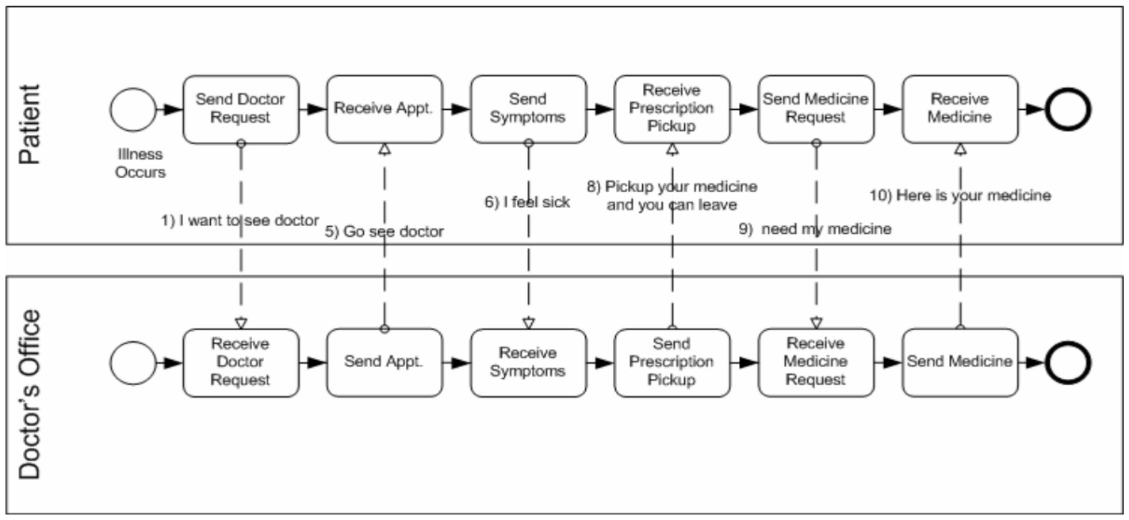
The corresponding population with respect to compound activity of the BPMN diagram then becomes:

```
Activity-Compound activity = {('Send RFQ','1'),('Receive Quote','1'),
('Add Quote','1')}

IsRepeat = {'1'}

Intermediate event = {'Timeout'}

Compound Activity-Intermediate event = {('1','Timeout')}
```

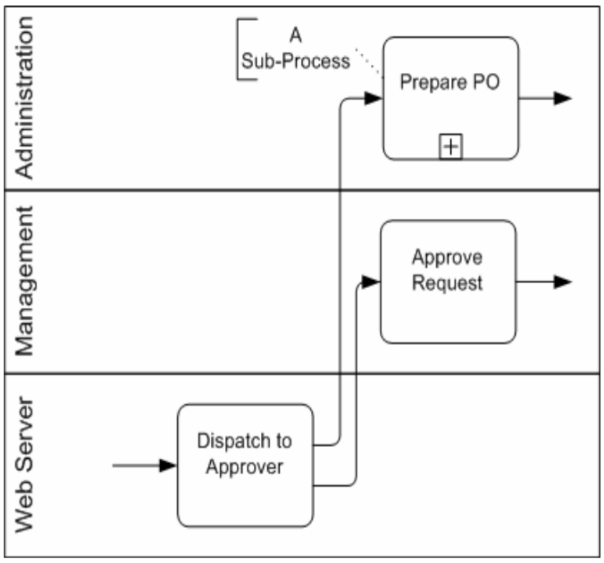
Since the above compound activity does not have a name, '1' has been chosen arbitrarily. However, the earlier proposal of the meta model and the one just listed does have the option for compound activities to be labeled. This is a consequence of a compound activity being a subtype of a flow object, which causes it to have a label through inheritance.

4.7 Defining the roles of pools and lanes in the BPMN meta model

The so called pool construct is introduced next and looks as following in BPMN form:



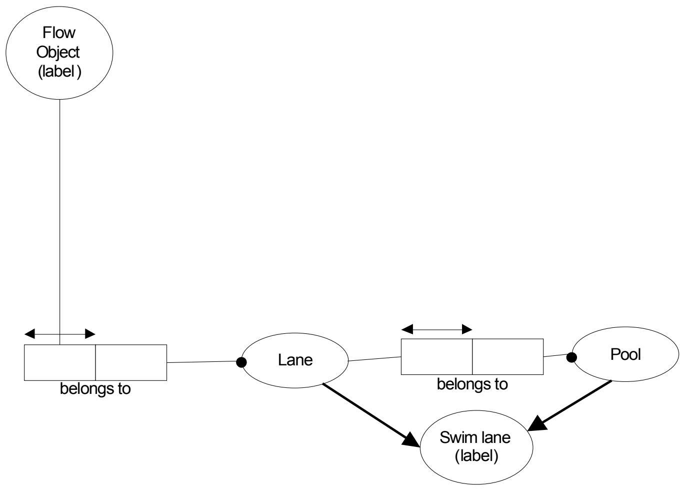
In order to introduce pools rightly, it is required to look at lanes first. Lanes share some properties with pools and are described as being a partition of pools. They are introduced in the following fashion BPMN-wise:



The message flow is covered by the connection between a flow object and flows as introduced before. This means the choice has been made to make the message flow and sequence flow indistinguishable. This is however not completely true, because by checking whether activities are in the same pool or not, one can determine whether one is dealing with a message flow or sequence flow. Making them indistinguishable makes sense, because they both express a sequential dependency and also because a data layer, that describes data dependency, will be introduced in the next chapter.

What needs to be stored in addition to what has already been mentioned, is that a pool consists of one or more lanes and that a lane in addition to that can contain one or more flow objects. Everything that is connected to the corresponding flows objects, such are artifacts, can then be contributed to belong to that lane automatically.

An ORM translation then becomes:



What remains then is to describe what population the two previous BPMN diagrams lead to. For the first one with the doctor-patient relationship the result is:

```
Flow-Flowobject(Out)={('1','Send  doctor  request'),('2','Receive  doctor  request'),
('3','Send appt'),('4',Receive appt',...}

Flow-Flowobject(In)={('1','Receive  doctor  request'),('2','Send  appt'),('3','Receive
appt'),('4','Send symptoms...}

Flow-Lane={('Send doctor request','Patient'),('Receive doctor request','Doctor'),('Send
appt','Doctor'),('Receive appt','Patient'),...}

Lane-Pool={('Doctor','Doctor's office'),('Patient','Patient's home')}
```

Note that the annotations next to the message flows have been omitted and that from the given population of Flow-Flowobject(Out) and Flow-Flowobject(In) the distinction between message flows and sequence flows is not visible as mentioned before. For the previous BPMN diagrams (which for a start might be a subset of a BPMN diagram), which did not explicitly contain lanes and/or pools, it still becomes possible to identify a default lane and pool. This in order to make sure that lanes and pools are part of every BPMN diagram.

The population of the final diagram is done in a similar way, but since no pools are mentioned here explicitly the just noted default pool is used. This results in:

```
Flow-Lane={('Dispatch to approver','Web Server'),('Approve request','Management'),
('Prepare PO','Administration')}

Lane-Pool={('Web Server','Default'),('Management','Default'),
('Administration','Default')}
```

4.8 Discussion about graphically reconstructing the BPMN meta model

So far a meta model has been constructed, which depends on what one sees when one look at a BPMN model. This has been done by classifying what one sees when looking at a BPMN diagram and finding a higher meaning, so that a generic meta model can be constructed. Ideally when this meta model is available with a corresponding population, it should be possible to reconstruct the BPMN diagram as it originally was graphically. To some extend this is possible by using everything that has been constructed so far, but when the exact positions in the two dimensional diagrams start to play a role, the current approach might not turn out to be sufficient yet. This paragraph discusses what is needed to reconstruct the original BPMN diagram graphically, but the concepts as discussed will not be added to the meta model, because they are not needed for the implementation.

There are two ways to reproduce the BPMN graphically of which one does not need extensions to the meta model as created so far and the other one does. It is even possible to have a combination of both approaches, but since they are extremes they are both given down here in order to give an impression:

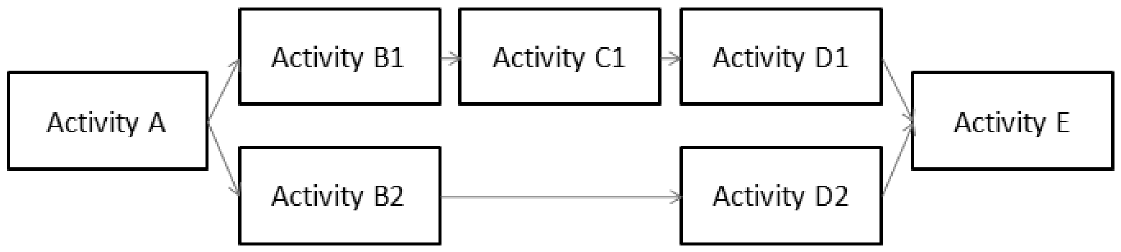
- Artificial intelligence approach: Using a smart algorithm that takes the population of the meta model as input and processes it and uses some rules of thumb to iteratively generate the BPMN diagram.
- Exact position approach: By extending the meta model as given so far with some graphical meta information (such as positions and dimensions), the position of objects is exactly defined.

Some guidelines that can be used using the first approach and properties that can be recognized in all the BPMN diagrams so far are:

- All objects of a certain kind have an equal size
- The sequential order of the activities that take place is ordered from the left to the right, with the objects that are shown most left taking place first
- There is no or little variation in the fonts used

It might be possible to think of reasons why these guidelines might not hold, which advocates the exact position approach:

- The size of objects might give some informal meaning. For example, activities that are drawn bigger in a BPMN diagram could be drawn that way in order to informally indicate that this task is more work then another task
- Sometimes a sequential order from top to down might be preferred in order to indicate sequential time
- An AI approach might have unpredictable results (Eg a small change in the population might change the diagram drastically, which makes it hard to compare two diagrams that are almost the same)
- In parallel diagrams, the vertical position of tasks might be used to indicate that they happen at the same time in normal circumstances (see example diagram)
- Some concepts might be hard to automate, because they depend on cosmetics or human judgment a lot, such as:
  - Not wanting to have crossing lines.
  - Activities that are in the same group should be close to each other.
  - Annotations should be in place in which they do not interfere with the readability of the business process too much.

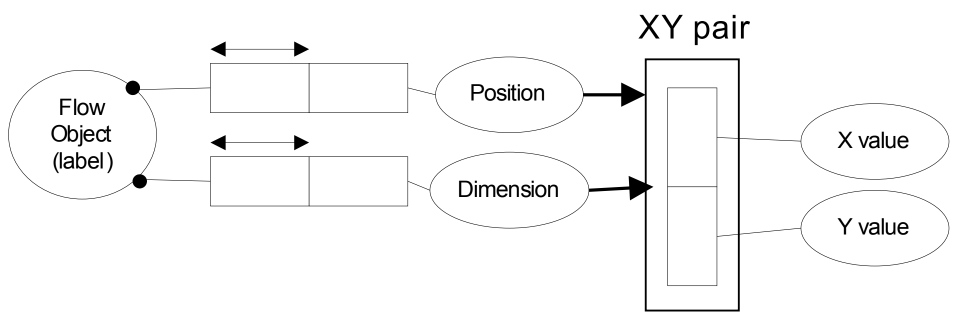


When the names of the activities are ignored, the positioning of the activities can be read to informally mean that activity B1 and B2 happen at the same time and that activity D1 and D2 happen at the same time. An AI algorithm (or any other algorithm) would not able to reproduce this without extra meta information.

Note that in the implementation that is given later in this thesis, the graphical information is not used, just like annotations will be ignored for the concrete implementation. In order to make it possible to reproduce the BPMN diagram as accurately as possible, the extension to make it possible to do so, is still given down here.

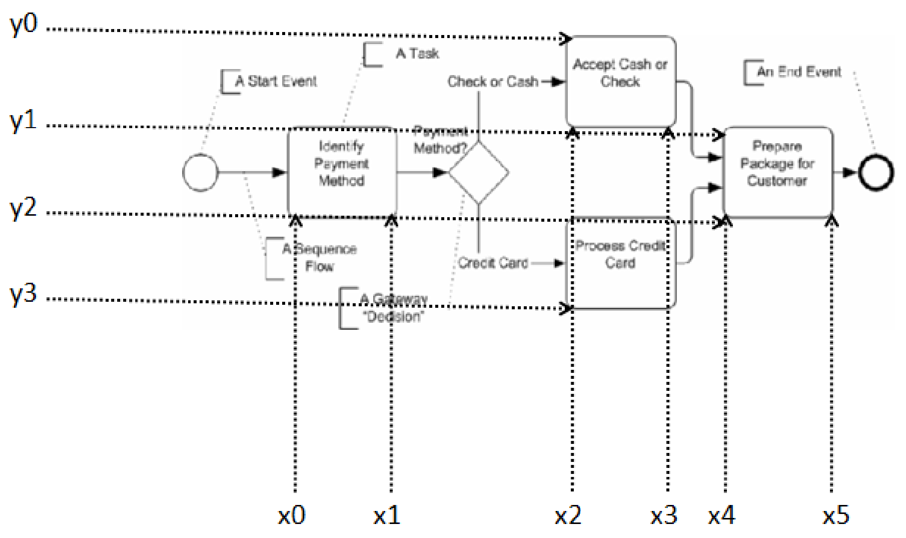
This extension also makes it possible to write a BPMN editor software product, in which the different elements of the BPMN diagram can be repositioned in any way the user pleases.

The extension will focus on flow objects only and the assumption here is that connections between them (which are connecting objects) are straightforward when the positions of the flow objects are known (additionally one could add so called elbows to connecting objects). Besides the position of these objects, the size they have (indicated as dimension) is stored. The results into the following meta model:



Two uniqueness constraints and two total role constraints have been added in order to make sure that every flow object has exactly one position and one dimension.

A position and dimension are added to all flow objects, which are each a subtype of an XY pair. Note that the X value and Y value not need to have a unit, since they are only relative values. The XY values could be defined in the following way, when the first BPMN diagram is used:



This leads to the following population:

```
Flow object-Position={('Identify payment method',(x0,y1)),('Accept cash or check',
(x2,y0)),('Process credit card',(x2,y2)),('Prepare package for customer',(x4,y1))}

Flow object-Dimension={('Identify payment method',(x1-x0,y2-y1)),
('Accept cash or check',(x3-x2,y1-y0)),('Process credit card',(x3-x2,y3-y2)),
('Prepare package for customer',(x5-x4,y2-y1))}
```

Note that for simplicity reasons the position and dimension of the gateway, start event and end event have been omitted in the above example. With them being part of the flow object population, they have to of course be included. In the population of the dimension, entities contain the minus symbol. This refers to the mathematical subtraction, since the dimension of a flow object is the difference between its coordinates on the corresponding axis.

What has been mentioned before is that this extension is not going to have consequences for the implementation. One is of course free to decide differently and by having the population of the extension at hand, one could decide to have the position and dimension of the flow objects have effect on the implementation.

4.9 Constraining the population of the BPMN meta model

So far small ORM models have been given, which are a consequence of real examples in the BPMN language and simple construct as described in an introduction paper. These models contain overlap, which makes it possible to combine them and have one all encompassing model. However, before this is done it is a good idea to constraint the population of the model somewhat. This includes uniqueness constraints (which states that a certain instance can only appear once in a certain role), total role constraints (which states at least one occurrence or obligated occurrence) and more complex constraints.

For flow objects this means:

- A flow object should exactly have one incoming flow and at least one outgoing flow except for:
  - End objects: Which only have one incoming flow and no outgoing flows
  - Start and intermediate objects: Which only have outgoing flows
- A compound activity consists of at least one activity and all the activities can maximally belong to one compound activity
- A compound activity can be interrupted by one or more intermediate events, but every intermediate event is only related to exactly one compound activity (to avoid ambitiousness in what caused the event to trigger).
- A flow object can be part of any number or groups, but belongs to exactly one lane.
- A group is defined as a set of flow objects.

For connecting objects this means:

- They are either a flow which has exactly one input and one output flow object
- Or they are an association that connects exactly one flow object with exactly one artifact

For pools and lanes this means:

- A set of lanes is a partition of a pool, which means:
  - Every lane belongs to one pool
  - By taking all the lanes together that are part of one pool, one can have the definition of the pool
- Every BPMN diagram has at least one pool and one lane (if not just assign default ones)
- A lane contains at least one flow object, but every flow objects belongs to exactly one lane

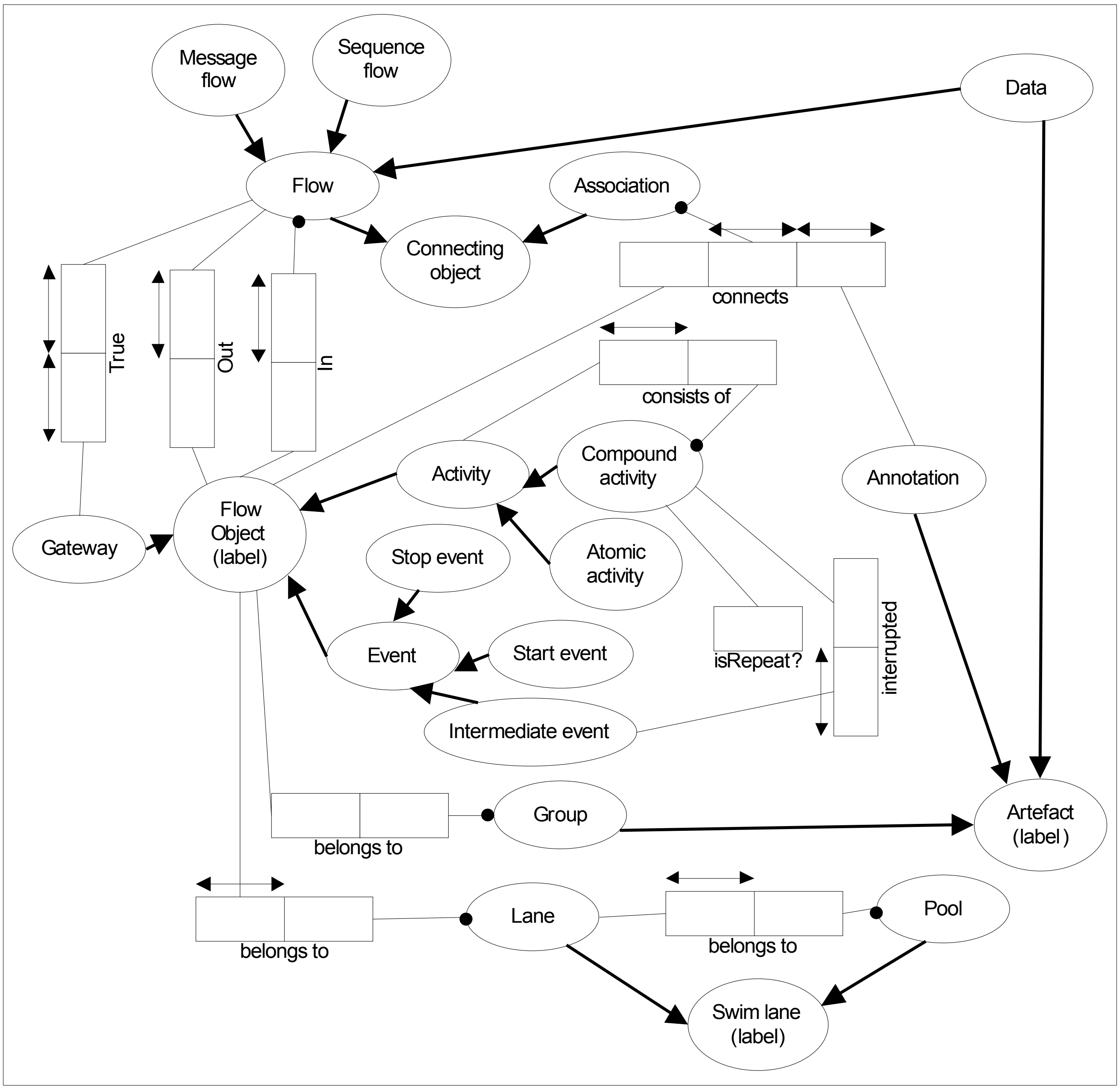
Some complex constraints that are about several entities:

- Every flow (a sequence of flow objects and connecting objects that are connected) should start with a start object and end with an end object, unless:
  - The flow is part of a compound activity
- A flow object should exactly have one output, unless:
  - The flow object is a gateway, which has one output for true and one for false, so two in total.
  - The flow object is connected through a connecting object to flow objects outside its lane. A maximum of one output to flow objects per lane is allowed then.

Several of the above constraints can be easily translated to ORM concepts, but other ones are hard to represent graphically and can only be checked by using mathematics. Therefore only the easy ones have been added to the model, which leads to the final model as shown in the next paragraph.

4.10 The full BPMN meta model

In this paragraph all the smaller ORM models as presented before have been taken together, after which some of the constraints of the previous paragraph have been added. The result is the model as indicated below, which serves as a formal meta model of BPMN:



4.11 Design decisions taken during the creation of the BPMN meta model

As can be seen in this chapter, the creation of the meta model has not been straightforward and when other people would have been asked to make a meta model, results might have differed on several aspects. Therefore this paragraph will make design decisions, that other people would have taken differently, more clear and will try to make it assumable why certain choices have been made. This does not mean that other choices might not have been satisfying as well.

First of all the language that has been used to model everything named ORM. ORM offers a formal basis to express objects and the roles they engage in. The leads to an unambiguous meta model which can serve as a good framework for both discussion and implementation later on. Alternatives that have been considered are UML and NIAM. UML might be more widely accepted for many modeling purposes, but does not always offer the completely formal foundation as needed. NIAM[31] is a predecessor of the ORM language, but puts more focus on the communication aspect then ORM, which is expressed in its excessive usage of label types. Since the intention here is to describe concepts ORM turns out to be more suitable. Compared with relational database models or XML representations, ORM offers a more conceptual view on the domain versus a more implementation driven alternative by databases and XML. It is possible to make a translation from ORM to relational databases or XML when needed in a structured automated way[53].

Second of all the subtyping of the four kinds of entities. The BPMN paper clearly distinguishes several kinds of objects. These have been translated as literary as possible in ORM and when kinds of entities share properties, subtyping is the solution that makes most sense.

Third the connection of most roles to the flow objects and flows. Within the flow objects and flows there are many different subtypes, which each show different behavior. It thus makes sense to connect many roles to the subtypes and show the subtype specific behavior here. However, when this is done, it turns out that there is much overlap among the subtypes or that the subtype hierarchy needs to be redefined. Instead of this the choice has been made to minimize the number of roles and to exclude some populations (eg a start event having an input) by having complex constraints. Unfortunately it turns out hard or impossible to make visible graphically, but that is the price that has been paid for a more simple model.

Fourth the removal of the option to connect annotations to connecting objects. The connection of annotations to flow objects is included to the meta model and is also visible in the BPMN example diagrams. The BPMN diagrams do however also show the option of connecting annotations to connecting objects. There are two reasons why this has been omitted from the meta model:

- A connecting object in contrast to a flow object, does not represent a real work of activity and is merely there to indicate a chronological order and dependence between flow objects. It therefore becomes hard to assign a real meaning to connecting objects
- If one would allow annotations to connecting objects, a recursive unwanted structure would become possible. One could use an association to connect an annotation to a flow object, but as a result annotating this association with a new association becomes possible. This can be repeated forever.

Fifth the usage of the four BPMN diagrams as presented. The four diagrams are part of the paper about BPMN and are supposed to illustrate BPMN as briefly and as clearly as possible. Copying these examples therefore seems a sensible thing to do, also because these standard examples might already be known by many people that deal with BPMN (since they can be expected to have read this paper). This improves the readability of this document.

Sixth the choice for the Boolean decision at a gateway. The following alternatives with their corresponding explanations have been considered:

- A different representation for two options then a Boolean: The Boolean has been chosen to illustrate the two options, with one option being true and one false. Instead of this a data type named 'direction' consisting of the options 'left' and 'right' might seem more illustrative at first. There are three reasons advocating the decision for using a Boolean though:
  - Simplicity: The Boolean is the most basic type one can imagine in computer science and it is often a good idea to go for the most simple solution.
  - Why bother?: The way things have been solved in the meta model is by having one role linking the 'positive' path and one linking the 'negative' path. The underlying data type does not really matter, as long as there are two options.
  - Data filter: As already briefly mentioned, the chapter dealing with the data layer will introduce a filter working on a data type in order to decide what path to take. Intuitively naming all the positives satisfies here, which resembles a Boolean very much.
- A multigate solution with integers: In this case a gateway could have several (1 or more) exit gates instead of two. Every gate would be numbered using a single integer value in the meta model population. By using the Boolean option one still has the ability two represent more then two options. Namely by putting different gateways in sequence and filtering out positives time by time. The integer solution can then be seen as syntactic sugar for a Boolean sequence, but does not increase functionality. In addition to this all the BPMN examples that have been shown only deal with a binary gateway which raises the question if more then two options are wanted at all. It might be a good idea to ask the business people first, whether a gateway with more then two options is even wanted.

Seventh the omission of some gateways and events. Appendix C of this documents shows that BPMN offers more gateways than the examples have been providing so far. The reason why this has not mentioned in the meta model so far, is that distinction of different gateways is seen as an artifact to clarify things of the implementation level. This makes it more of an implementation than a conceptual issue. The later chapter about the data flow meta model will show that by defining how the data flows, dependencies are enforced automatically. Additions to the model will be proposed then and by introducing extra complex constraints, a restriction on and a link between the data flow and the kind of gateway can be made then. For events something similar is the case. However, a distinction between the start, intermediate and end event has been made though, because of their essential differences.

4.12 Conclusion

A meta model for BPMN has been given in the ORM language. This meta model makes it possible to describe a BPMN diagram in terms of a population and makes it possible to reconstruct the diagam given a population. Also the model in the ORM language makes it possible to formally reason about BPMN diagrams.

While creating the meta model, four concepts turned out to be important, namely flow object, connecting object, artifacts and swim lanes. After they have been addressed, their connections with other objects (roles or fact types) have been investigated by providing BPMN example diagrams.

After this some extension has been mentioned, which stores positions and dimensions of objects in the 2 dimensional space, but this extension is not part of the final model, because it does not have effect on the implementation.

Finally some constraints and design decisions are mentioned, which lead to an overview of the final model. However, it turns out that for implementation purposes, this meta model is not sufficient. Therefore the next chapter will be devoted to an extension on this model in which decisions with respect to the implementation are made. The extension of the model is not visible graphically in the original BPMN model.

5 Extending the BPMN meta model

What becomes clear from the previous chapter is that BPMN offers a very intuitive style of displaying what a business process or workflow looks like. This even holds for people who are not familiar with computer science. When one starts to think about how to implement this previously shown diagrams however, it turns out more information is needed due to the informality of the diagrams. Therefore the decision has been made to extend the BPMN meta model in this chapter.

This extension has been made as generic as possible, but one cannot avoid anymore to take a look at the implementation level already. This means that decisions that are going to be made here, might be platform independent, but will in many cases turn out to be taken having a certain platform in mind. Also everything that is going to be decided here is not part of the graphical notation of BPMN anymore. However, it will turn out later that many concepts as represented here, can be graphically represented by adding an extra graphical layer to the BPMN diagram.

The actual implementation requires four additions to what has been shown so far, which are:

- Data layer: The diagrams so far only showed in what order activities are executed. This does not mean by definition that the data flows exactly the same way. Of course all data that is used later in the process, needs to be generated in an earlier step. But data that is generated early in the process, does not need to be used right away or at all, depending on what path the process is taking. In order to prevent that all the data is dragged around the whole process and going through all activities, the chapter will show an approach in which activities (but also other flow objects) are typed and in which inputs and outputs of activities are mapped to each other. The data layer aspect is explained in three paragraphs: One for atomic activities, one for compound activities and one for additions to the meta model with respect to gateways.
- Activities: These are the main and smallest building bricks of the BPMN diagram in which the actual work happens. However, the diagrams as shown before do not show what needs to be done at the smallest level and are in their current form submissive to a completely free to choose implementation. When a formal implementation is wanted which can be executed on a computer, activities need to be linked to some piece of software. As later in this chapter will be shown, this can be done in several ways, which need not to exclude each other. This does not mean that the ways that are presented in this thesis are the only ways to go. In fact, any way that matches the interface as described in the data layer section, can be plugged into the solution as proposed here.
- Triggers: Events are the connections of the process with its environment. For example when the start event is triggered, the process will start running. This trigger to the process is given for some reason (eg a new product has been sold) and the environment needs to have an option to indicate this. Also the process has the option to trigger the environment, for example through an end event, indicating that some work has been finished. Once again, just like activities, different approaches will be shown here which can complete each other and do not need to exclude each other.
- Resources: The concept of resources is very much related to the data flows. A resource is an entity that is available in one specific lane and no other lanes. This entity could be a database to store data, but could also be a (human) actor that can perform a special skill, such as:
  - Transforming input of type A to output of type B (this could represent making a smart decision based on explicit or tacit knowledge)
  - Performing an action in the real world (eg putting a letter on the snail mail)

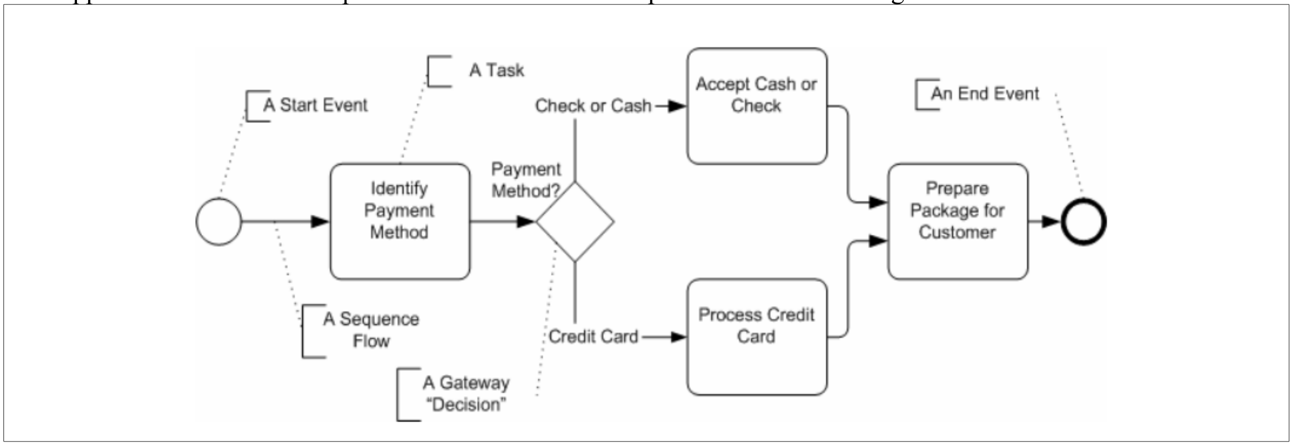
The just mentioned needed additions will be discussed in the next paragraphs and will involve additions to the meta model of the previous chapters. This means that when entities have the same name as entities in the previous chapters, the same concept is meant. This also means that it is possible to merge the model of the previous chapters and this chapter together and end up with one big model that can be populated. This is not done is this paper though, due to the enormous size this model would have and thus its inability to make it fit nicely on paper (read: to make it understandable).

5.1 Adding a data layer to the BPMN meta model extension

The data layer provides information about how the data flows and is implemented for atomic activities, compound activities and gateways.

5.1.1 Adding the data layer for atomic activities to the BPMN meta model extension

As noted in the introduction of this chapter, activities and other flow objects will be typed and have a certain input and output. In order to make a meta model for this, the BPMN diagrams which have been introduced before will be reused and supplied with a fictional implementation. Take for example the first BPMN diagram which looks as follows:



The process describes a sequence of activities that take place after an order has been made and this includes the payment and the preparation of the package for the customer. In order to make the actions paying and preparing the package possible, it makes sense to have the following data at hand even before the process is started:

– Order details: Which items have been ordered and what is the total price of the order?
– Customer details: For whom is the order meant and where does that person live?

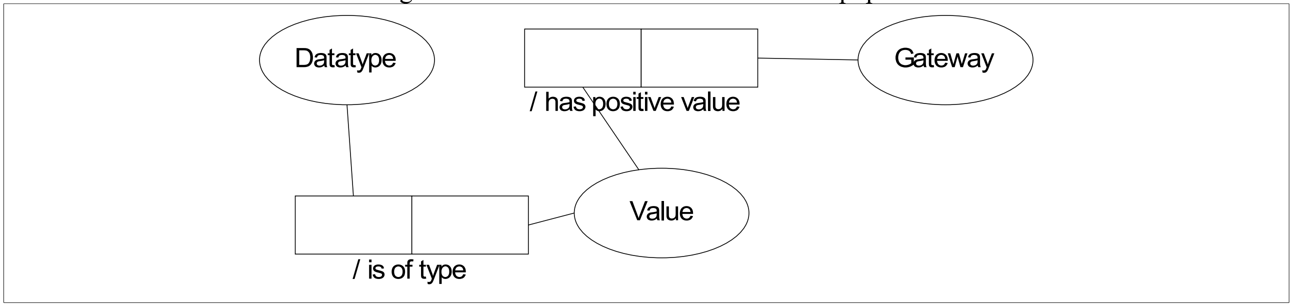
Note that the payment method is not needed as input for the above process, since it is determined during the process. In order to have information available at the beginning of the process such as order details, it is considered to be output of the starting event.

After the starting event is triggered (by some entity outside the process that wants to process to be executed for some reason), 'Identify payment method' needs to be executed. Since this activity does not need to have any information (unless for example orders greater then 500 euros can not be paid by cash), its input is empty. Its output is something of type payment_method.

The activities 'Accept Cash or Check' and 'Process Credit Card' need to have information about the order in order to obtain the amount that needs to be paid. Therefore two data flows from the starting event to both activities containing the Order Details are made.

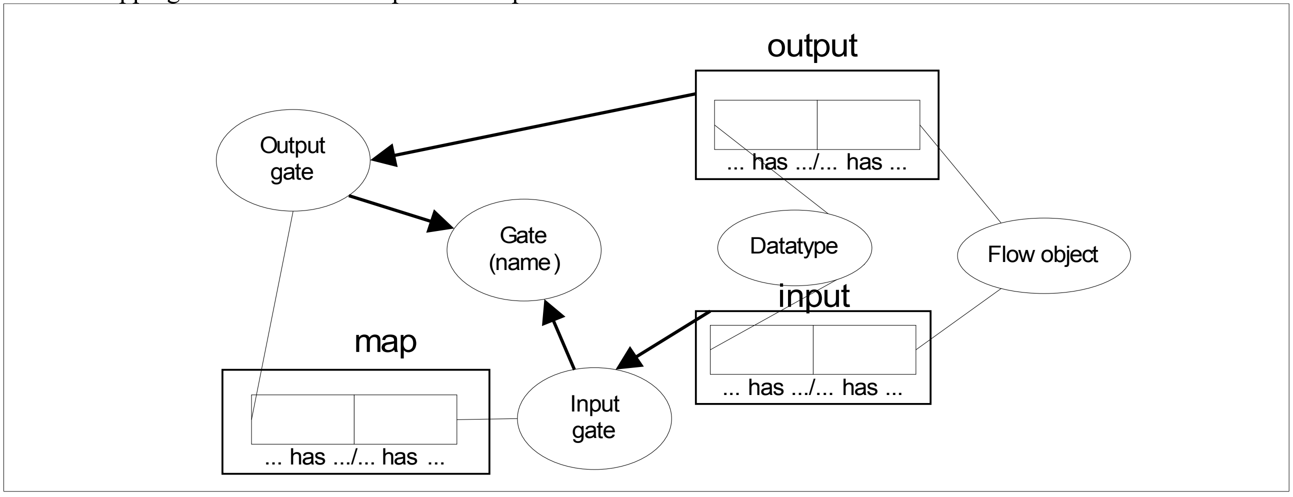
'Prepare package for customer' requires both the Order details (what to pack?) and Customer details (where to send it to?) as input data. This causes to data flow links between the starting event and 'Prepare package for customer'.

As mentioned before the gateway is something of type Boolean. In the above diagram 'Check' or 'Cash' lead to a certain activity and 'Credit Card' to another. The Boolean function could thus return true in case 'Check' or 'Cash' is the input and false otherwise (note that the order of true and false is arbitrary and could have well been reversed). In terms of the meta model this leads to the following for the Boolean function which will be populated later on:



The just shown meta model also makes it possible for every data type to define all the values that belong to it. Also there is the possibility that a value belongs to more data types (which might be handy for subtyping). However, in the implementation layer this option is purposely not available, in order to enforce strong typing. A short example population for defining types is included in the sample population later on.

For the mapping and definition of input and outputs the addition to the meta model is as follows:



Note that the input and output are not directly mapped to each other, but instead of this are the subtype of respectively 'Output gate' and 'Input gate'. Later when composition is discussed, it becomes clear why, but for now it is sufficient to state that this is done to anticipate on what is about to come.

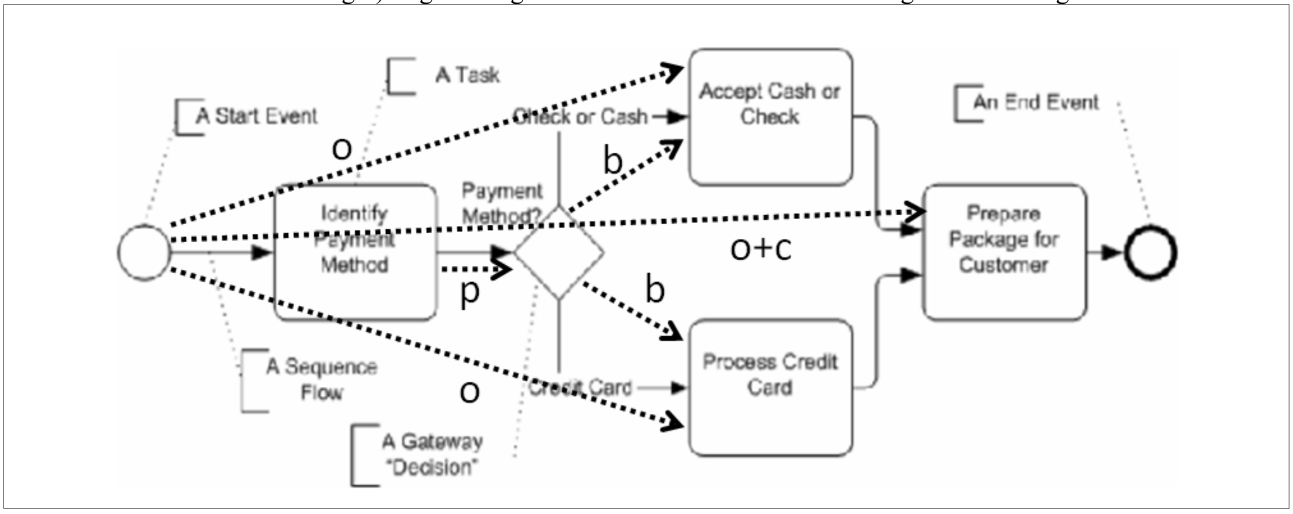
The population of the two meta models with respect to what has been mentioned so far is:

```
Output = {('','Order details','Order'),
          ('','Customer details','Customer'),
          ('Identify payment method','Chosen payment method','Payment method'),
          ('Payment method?','Gateway output','Boolean')}

Input = {('Payment method','Gateway input','Payment method'),
         ('Accept Cash or Check','Order input','Order'),
         ('Process Credit Card','Order input','Order'),
         ('Prepare Package for Customer','Order input','Order'),
         ('Prepare Package for Customer','Customer input','Customer')}

Gateway-positive-value = {'Cash','Check'}

Value-datatype = {('Cash','Payment method'),('Check','Payment method')}

Map = {(('Identify payment method','Chosen payment method','Payment method'),
         ('Payment method','Gateway input','Payment method'))
       (('','Order details','Order')
         ('Process Credit Card','Order input','Order')),
       (('','Order details','Order')
         ('Accept Cash or Check','Order input','Order')),
       (('','Order details','Order'),
         ('Prepare Package for Customer','Order input','Order')),
       ('','Customer details','Customer'),
         ('Prepare Package for Customer','Customer input','Customer'))}

Datatype-Value = {('Boolean','True'),
                  ('Boolean','False),
                  ('Int',0),
                  ('Int',1),
                  ('Int',-1),
                  ('Int',2),
                  ('Int',-2),...}
```

Note that '' stands for the start symbol, which is not labeled. Order and customer are the data types representing the customer details and order details respectively. The middle role of the triples represents the gate name. The map looks a bit sophisticated in the above population, but simply only contains tuples that connects the output gates of activities to input gates of activities.

In addition to the Boolean data type the integer data type has been added for illustrational purposes. Note that this is just a conceptual view on the domain and that actually storing all possible integers on the computer would be a little too far fetched.

Since the previous population might not be straightforward to read, adding an extra layer to the diagram (which can be either shown or not to the manager) might be a good idea. This leads to the following extended diagram:



The following abbreviations have been used:

- O: Order data
- C: Customer data
- B: Boolean
- P: Payment method

The O+C stands for two data flows, which are in the above case represented by one single arrow for simplicity. When desired it is possible to add the names of the gates as well, which is not done in the above diagram due to a lack of space.

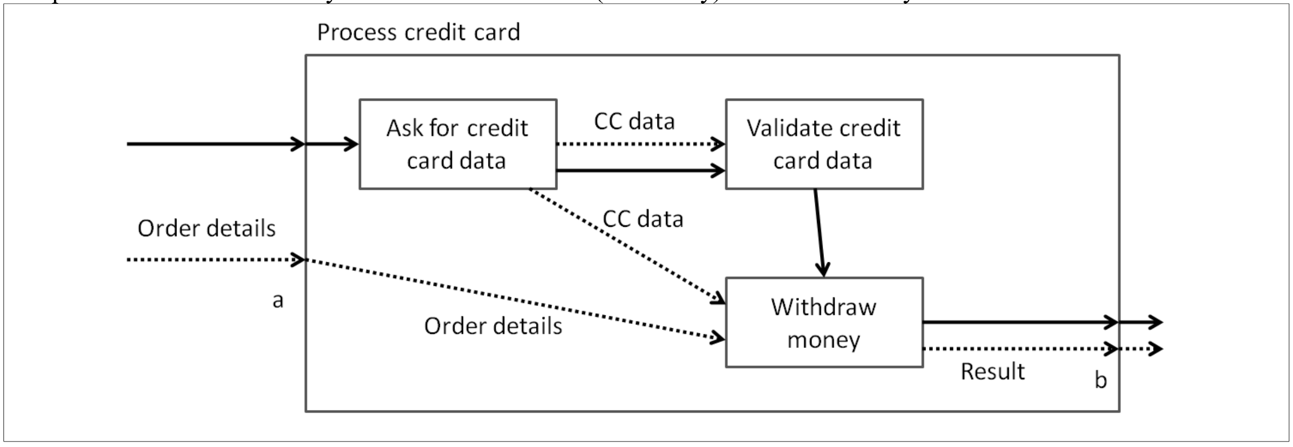
5.1.2 Adding the data layer for compound activities to the BPMN meta model extension

For compound activities two approaches will be taken, with either one having their positive and negative properties. In terms of implementation they might turn out to be the same however, but conceptually there is certainly a difference. There will be an approach in which the compound activity is embedded as a special activity in the current business process model. The second approach deals with a compound activity being a business process on itself, which as a result can be embedded in a new business process as if it were an atomic activity. The former approach is more white box based and the latter one stimulates a black box approach. The next table summarizes the differences and the later designs of both ways will show what is meant more carefully:

Compound activity being part of current business process model	**Compound activity as a new business process**
- White box approach - Everything can be put in the current model, which only deals with one business process - Re-usability is not straightforward - Intermediate results from the compound activity are accessible - Compound activity can cross lane and/or pool boundaries, due to context dependency - Easier support for repetition in some cases	- Black box approach - More then one business process needed, while the model as introduced so far only supports one - Encourages re-usability - Small changes might need a lot of redesigning - Encapsulation - Context independent - Intermediate results can not be accessed from the outside process

Since both approaches have their distinct properties, which can be seen as advantages or disadvantages, that do not exclude each other, a design for both approaches will be shown. Recall the BPMN diagram from the previous paragraph (about atomic activities). This diagram shows an compound activity named 'Process credit card' which in that paragraph was considered to be an atomic activity. The input of this activity is the order data (to obtain the amount to be paid), while no output was needed (in case the payment is unsuccessful the process aborts or stalls, which causes the last activity in the BPMN named 'Prepare package for customer' never to happen as most businesses would like to be case).

The process credit card activity has been differentiated (fictionally) in the follow way:

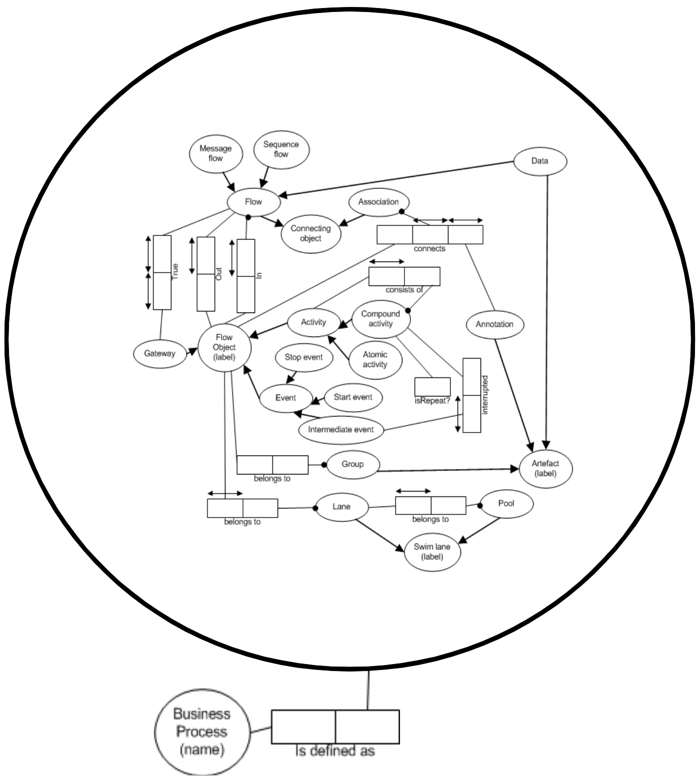


The above diagram can be read as follows: In order to Process someone's credit card, three steps need to be taken, namely 'Ask for credit card data', 'Validate credit card data' and 'Withdraw money'. The order of the steps is indicated with the arrows. The dotted arrows on the other hand, indicate data flows. For example, 'Withdraw money' requires the credit card data (in order to determine the bank account to withdraw from) and order details (in order to determine the amount to withdraw). Note that the sequential flow has been connected with the input and output with the solid sequence arrow. The normal convention in BPMN is to not showing the beginning and ending connections and to just make it intuitively visible where the compound activity is supposed to start. The letters a and b are markers which will be used in the white box approach later on.

First the black box approach, since its more simple. The meta model as discussed and developed so far is meant to model one business process. There are two ways to deal with an extra business process:

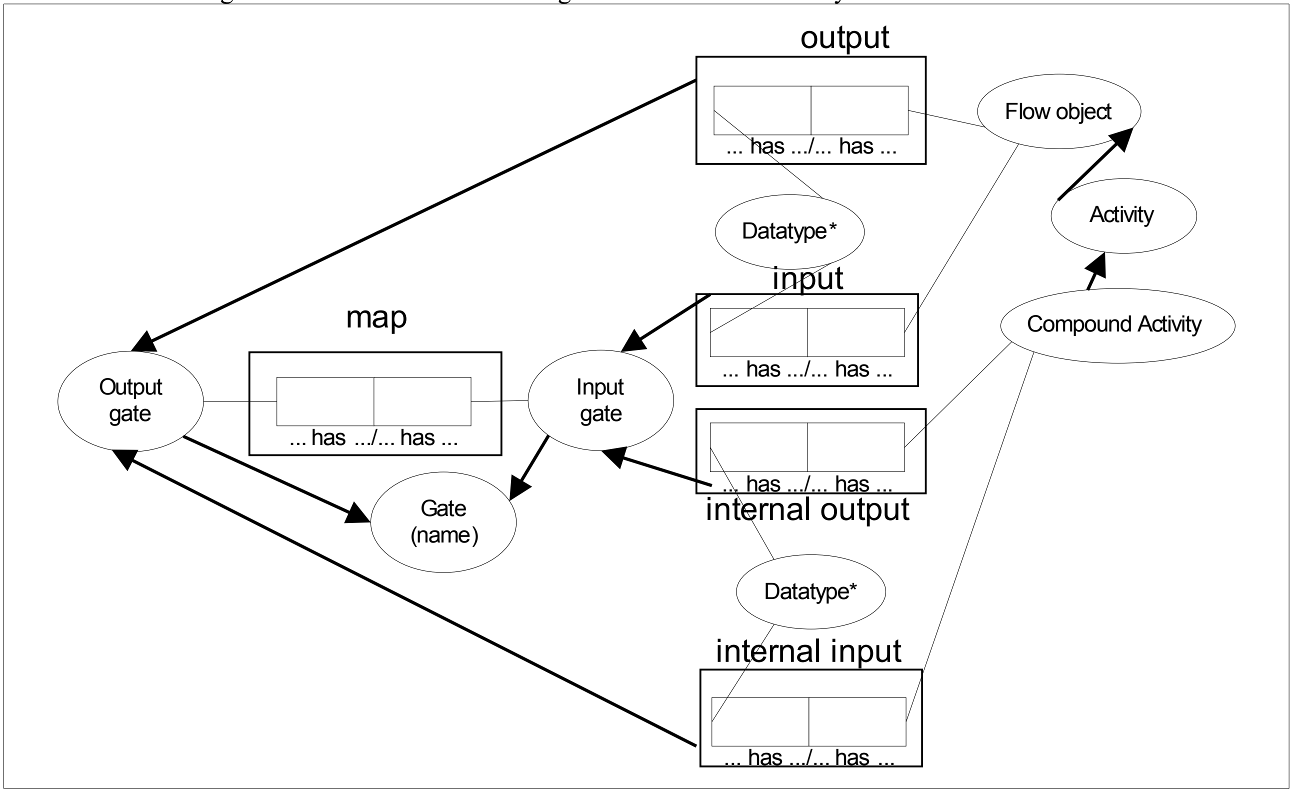
- One can decide to make a new population for every single business process that ever needs to be made and just keep them strictly separated.
- One can embed this possibility in the meta model by indicating for every entity of the model to which business process it belongs.

For simplicity reasons the first approach will be used here, which results into the case that whenever more business processes are used, several populations (one per business process) will be given. This leads to the following meta model (to which later the data layer and everything that follows need to be added):



Second the white box approach which embeds the compound activity in the meta model that is already there. It turns out that in order to do this, two concepts are missing. They will be called internal input and internal output here. What is meant by them is the gates that connect what is coming from and going outside (as marked with A and B in the previously shown diagram about 'Process Credit Card'). Note that the internal input gate is an output gate (marked with A) and the internal output gate (marked with B) an input gate. Be aware that this might lead to confusion occasionally.

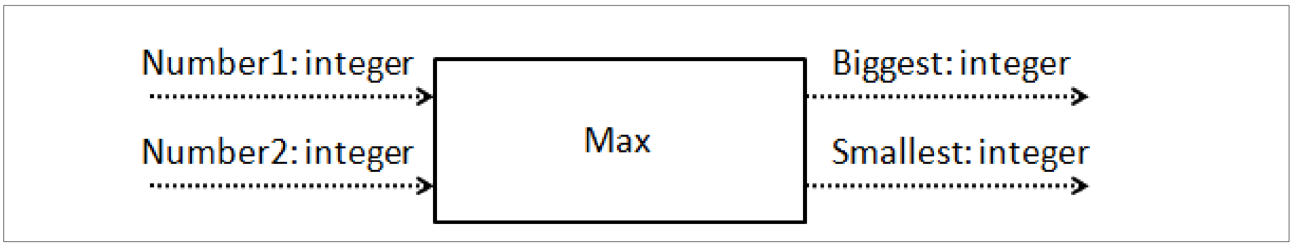
When the two extra gates are added to the ORM diagram of the atomic activity the final result becomes as follows:



The corresponding population with respect to the 'Process Credit Card' activity as proposed is then:

```
Flow object-Output gate={('Process credit card',0),
                         ('Ask for credit card data',1),
                         ('Withdraw money',2)}

Flow object-Input gate={('Process credit card',100),
                        ('Withdraw money',101),
                        ('Withdraw money',102),
                        ('Validate credit card data',103)}

Compound activity-Input gate={('Process credit card',200)}

Compound activity-Output gate={('Process credit card',300)}

Input gate-Output gate={(0,300),(100,200),(102,1),(103,1),(300,101),(200,2)}
```

Note that for the gates labels numbers has been used for now, but as the above meta model indicates every gate can have a name. The reason why it is very wanted for gates to have names is illustrated by the following function that takes the maximum value of two numbers:



Without gate names it would not be very clear what is going on, since the data type of all input and output gates is the same.

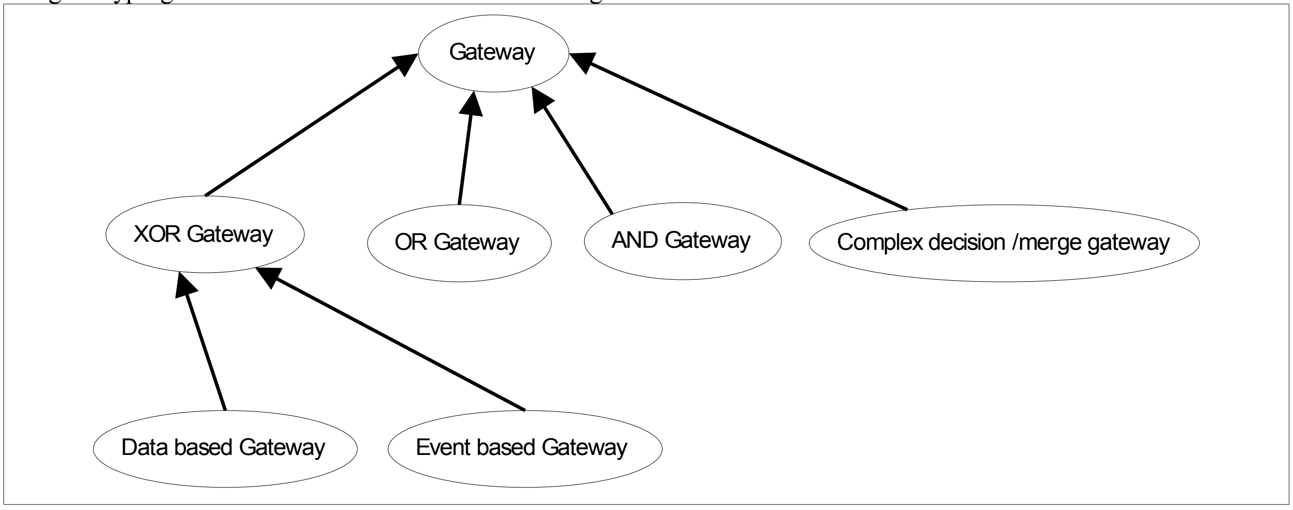
So far a match between the data types in a map has not been enforced (which need to be done for both atomic and compound activities). Since it would be complex to express this graphically, it is part of the complex constraints at the end of this chapter. Initially data types have to match one on one, which means that whenever a map is done both the input gate and the output gate should have the same data type. There are however two exceptions to this rule, namely:

- Subtyping in data types: When in a map one gate depends on another, it is satisfying when the depending gate has a superset of values in its data type. This is the case because the depending gate can at least deal with all the values it can expect. (For example: When a activity expects a credit card or cash as payment method, it is not a big deal when it is fed with cash only. As noted earlier this is explicitly forbidden in the implementation chosen in this thesis, in order to enforce strongly typing, but might be considered when the typing system supports subtyping in the way as just described.
- Repeat compound activity: When the compound activity is a Foreach operator, it should be fed a set of data values, of which it picks one at a time until the loop is done. For example: The compound activity named 'Check credit rating for all customers' which is a repeat activity, gets a set of customers. What happens internally is that only one customer is being dealt with at a time. This means that the input gate is of data type set of customers, while the internal output gate is of data type customer. Although they are not of the same data type, they are mapped. Even worse, if they would be of the same type, the map would make no sense in the repeat case or would perhaps mean repeating it for one instance.

5.1.3 Adding the data layer for gateways to the BPMN meta model extension

Appendix C offers an overview of events and gateways that purposely have not been added to the meta model so far, because they are directly linked to the data layer meta model. This paragraph will show the extension to the meta model that is needed to deal with these special gateways, while the paragraph dealing with complex constraints later in the chapter will show the consequences for picking special gateways on the data layer model.

Since the newly introduced gateways are special kinds of the general gateway concept, it makes sense to express this by using subtyping in ORM. This results into the following addition:



5.2 Adding different implementations for atomic activities to the BPMN meta model extension

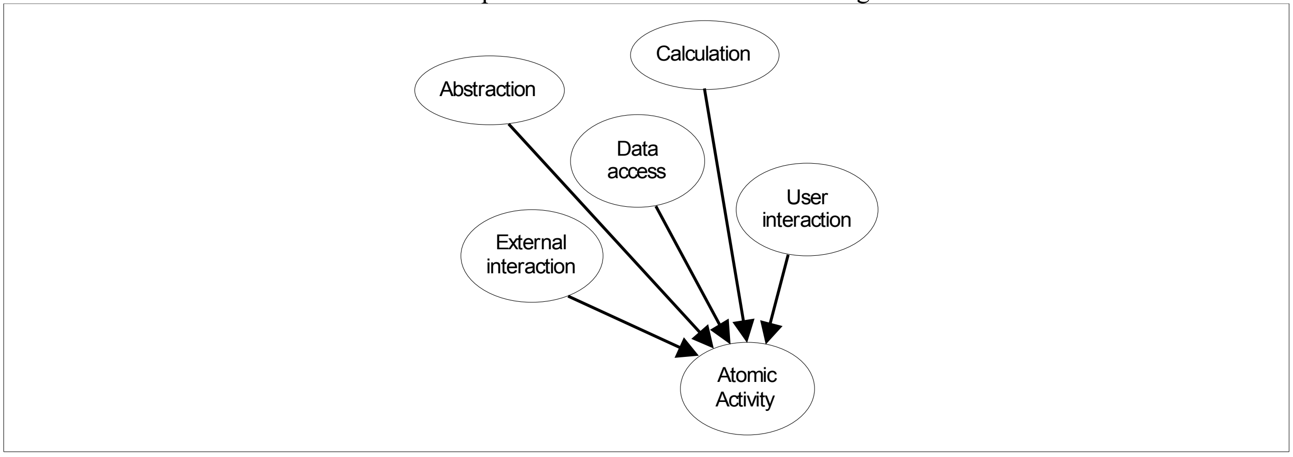
Atomic and compound activities have been discussed so far in the previous chapter. At the lowest level of an activity (read: the most atomic activity possible), 'real' work needs to be done. Five options have been selected here, which will be implemented in the next chapter using the Itasks library on the Clean language platform. The five options are supposed to represent the most common ways in which an atomic activity can be implemented and their concrete implementation is just meant as an concrete example.

Remember that an activity is defined as some unit having a number of input gates and output gates, each with their own specific data type. Every option has to satisfy this condition. The examples shown down here are very specific examples of what is possible in order to make the implementation possible, but the general idea remains that input is transformed into output. One is therefore always free to make additions to the five options as presented here or neglect some of them for certain reasons.

The options and their underlying ideas are:

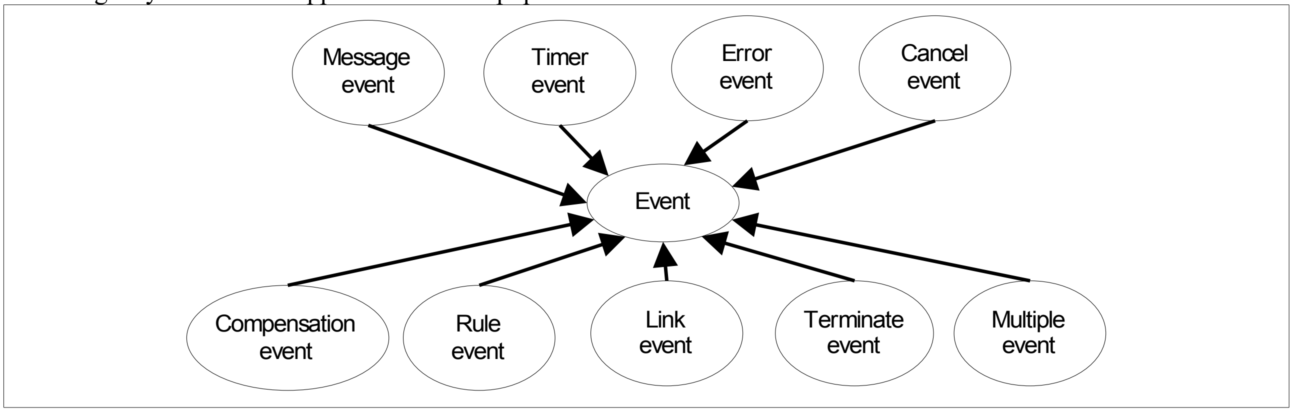
- Calculation: For every output gate a mathematical function without side effects is defined, which can use any information of the input gates as parameters. An example for the 'Validate Credit Card' function is the function 'creditcardnumber%11==0' which takes the creditcardnumber of type 'integer' as input gate. The output gate named 'valid' is of type 'Boolean' and will only result in true when the credit card number is dividable by 11.
- User interaction: The input gates are shown to a human actor, after which the actor has to enter the values of all the output gates. This means a human is selected to be the transformer between input and output. An example activity 'give feedback to this thesis' might have a text document and name field as input gates. This information is shown to the supervisor, after which the supervisor has to fill all the output gates (which are likely to be criteria on which a thesis is judged in this example).
- Data access: This option is similar to a user prompt, but instead of a human actor, a data resource is accessed. The input gates are used as a look up key, after which the retrieved data is written to the output gates. The distinction between humans and data resources turns out to be handy later when pools and lanes are introduced.
- Abstraction: In order to finish an atomic activity, a whole business process is executed. The input gates are linked to the starting event, while the output gates are connected to the stop event. This means that the transformation requires a sequence of transformations which are wrapped in a business process.
- External interaction: A function on the internet (also known as web service) is triggered and a wait for the result occurs. This makes it possible to let virtually anyone or anything in the world implement an atomic activity. For example: One could ask the credit card company whether the credit card exists in their records, after which a Boolean response as in the mathematical function example occurs. One tricky thing about using web services is that they are normally text based. This means that every input and output gate is of data type 'String'. In order to deal with this conversion functions named fromString and toString have to be defined for every output gate respectively input gate which respectively have signature (String->x) and (x->String), when 'x' is the data type that needs to be converted.

In the meta model the addition of the five implementations leads to the following result:



5.3 *Adding triggers to the BPMN meta model extension*

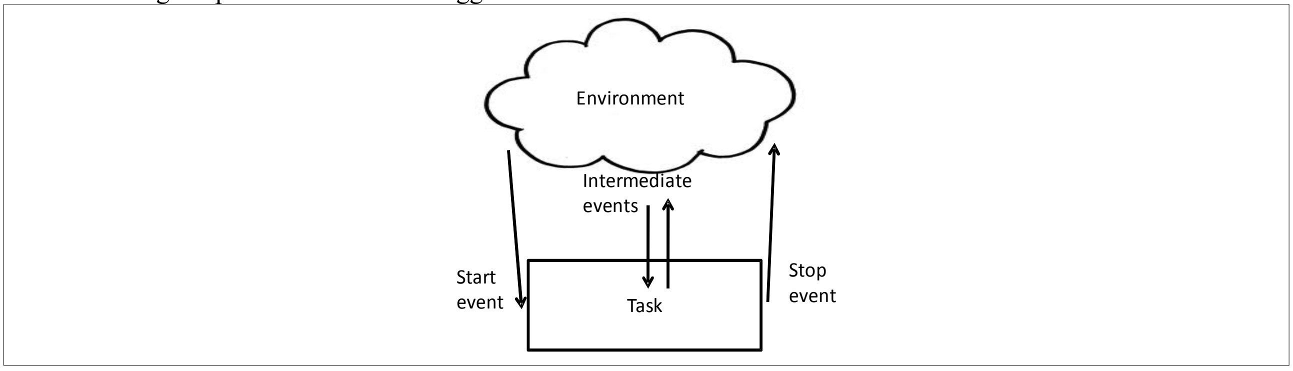
As the second half of Appendix C shows, BPMN offers a variety of events. The meta model can be extended in the following way in order to support them in the population:



Keep in mind that a start, intermediate and stop event have already been defined before. All the events that are shown in Appendix C can be obtained by combining the start, intermediate and end dimension with the dimension as shown in the above graph. More on this in the paragraph about constraints in this chapter.

Events are very much related to the concept of triggers and the different event types play the following role with respect to triggers (see diagram below):

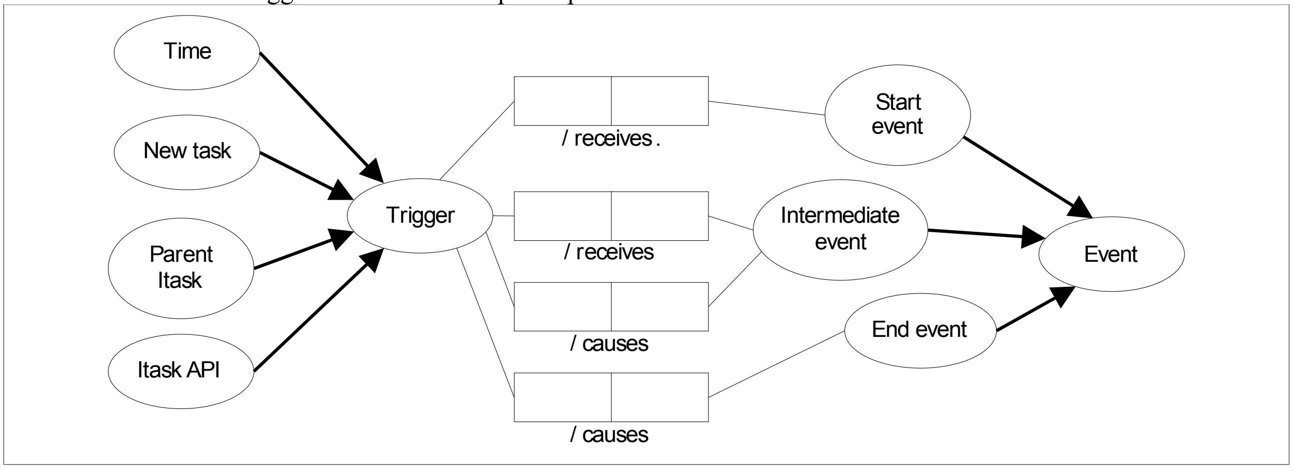
- Start event: Something from the environment of the process wants the activity to be started. Normally this leads to a new instance of the process. The process thus receives a trigger from the outside world.
- Stop event: After the process has ended the environment is notified in some sort of way (Eg the result of a calculation of the process can be used by another activity). The process thus causes a trigger in the outside world.
- Intermediate event: Can play a role that is similar to a start event or stop event. The main difference with them is however that the process is not initiated respectively ended by this event. This means these events happen during the process. In terms of triggers this event can thus either receive or cause one in the outside world.



The triggers are chosen to be of the following kinds at the implementation level:

- Time: A certain time has expired and causes an incoming or outgoing trigger
- New task: A new task is created in the Itasks GUI
- Parent task: A task is started by a parent task or a task has ended giving control to its parent.
- Itask API: The Itask system is approached from the environment

The term task is used in Itasks but represents something similar as an activity in BPMN.
In the meta model the trigger event relationship is expressed as follows:



5.4 *Defining the concept system and resources in the BPMN meta model extension*

In the BPMN paper the concepts lanes and pools are introduced, which are both a subtype of a swim lane. Different activities can share a lane, be in different lanes and be in different pools of each other. It does however not need to be possible for a single activity to take place in different lanes or different pools, since the BPMN examples as shown did not provide this concept. In a real life swimming pool, having different pools means having different water, while lanes that are part of the same pool share water but are not overlapping location wise.

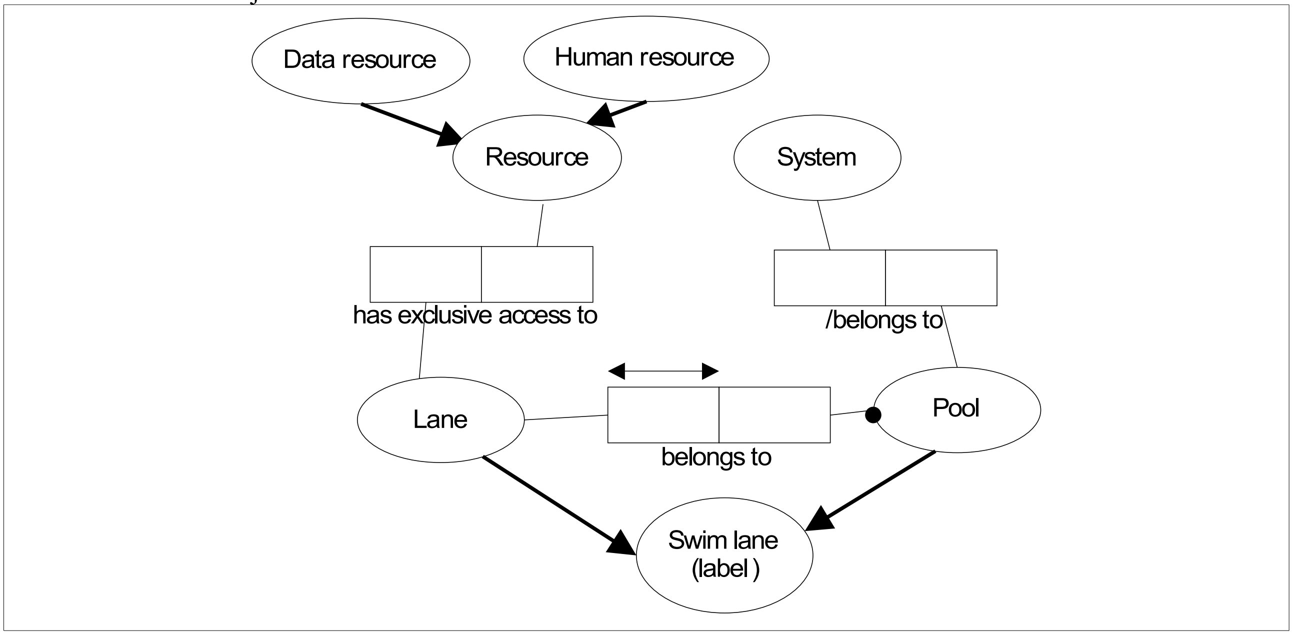
In the implementation that is proposed here the three possible relationships between activities are expressed as follows:

- Different pools: The activities take place in a different system and can only share information through messaging through an interface they have agreed upon.
- Similar pools, but different lanes: The activities take place in the same system, but have some exclusive resources which they but the other activity can not access.
- Similar lanes and pools: The activities have no secrets for each other and when one activity has a certain capability, the other one does automatically too.

In the above explanation two new concepts are introduced, namely resource (which is made more specific using two subtypes for illustrative reasons, but certainly can be enhanced with more options) and system, which can be defined as follows:

- System: A concrete implementation which translates a pool of a BPMN diagram to something that can really be executed (often being a piece of software).
- Resource: An actor that has certain capabilities in terms of transforming input to output
  - Human resource: A resource element that is played by a real human
  - Data resource: A resource element that encompasses a data collection. An element from this collection can be retrieved by given a certain input key

The meta model of the just mentioned looks as follows:



5.5 *Constraining the population of the BPMN meta model extension*

Everything mentioned in this chapter has been translated to an iterative addition to the meta model. Before the result of combining all these small models can be shown, it is valuable to add constraints first in order to limit the number of possible populations. In a similar way as the previous chapters, constraints are introduced in this paragraph, of which some are too complex to express graphically. They will still be mentioned here, but are not visible in the final meta model in the next paragraph. The constraints for this chapter, organized per category, are as follows:

Data types, values and gateways:

- A value belongs to exactly one data type (to enforce strong typing)
- A data type should contain at least one value (in order to make sure there is always something to work with at for example gates and gateways)
- If a gateway has a positive value of a certain data type, this gateway is dedicated with dealing with this data type. This means that all the positive values have to be of this data type. There can however be more gateways that deal with a certain data type. When the gateway does not have any positive values its data type is undefined and needs to be determined by its context.
- Gateways can only be of exactly one subtype of the overview shown in Appendix C (Eg a start event of type cancel is not possible). After one of the gateways is selected, the data flow should justify the selection of the gateway.

Input gates, output gates and mapping:

- When an input gate and output gate are mapped:
  - Their data types should match (except for the Foreach activity. In that case the input gate should be the set of the internal input gate).
  - Their sequential order in time should allow this mapping (eg it is not possible to map the output of activity B to the input of activity A when the latter is always executed first).
- When two gates are either both input or output gates of the same flow objects, they should have different names or they are the same gate.
- One input gate can only be mapped to exactly one output gate. This prevents ambiguity. An output gate however, can be mapped to any number of input gates.

Implementation:

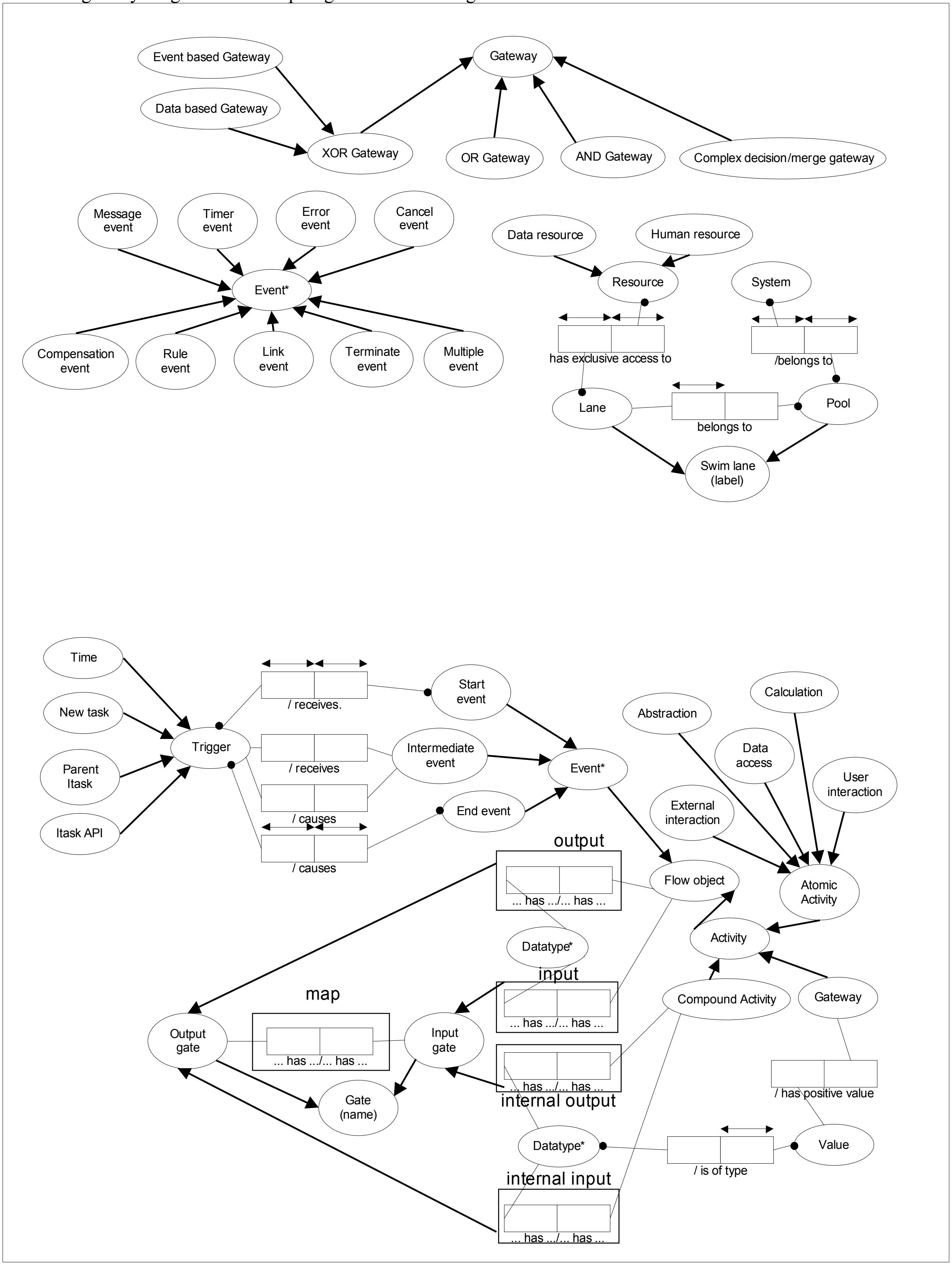
- Atomic activities have to have exactly one way of implementation. This in order to avoid ambiguity and to make sure every atomic activity is implemented at all.
- Triggers need to implement exactly one subtype in two dimensions. This needs to happen in such a way that exactly one trigger in Appendix C can be attributed to every trigger.
- Every event is assigned to exactly one trigger. This in order to make it possible to read from the BPMN diagram how many and which triggers play a role. For an intermediate event this means that it either causes a trigger or receives a trigger (XOR).

Swim lanes, pools and lanes:

- A resource is linked to exactly one lane. This is namely the chosen concept of a lane.
- A system encompasses exactly one pool. This is namely the chosen concept of a pool.
- When the implementation of the system is different then the one described here, its behavior should at least be similar to the one described here from the black-box point of view. This means that the other system can have a different implementation for atomic activities and thus differ in the meta model on this and this only.

5.6 *The full model of the BPMN meta model extension*

Combining everything from this chapter gives the following extension to the BPMN meta model:



5.7 Design decisions taken during the creation of the BPMN meta model extension

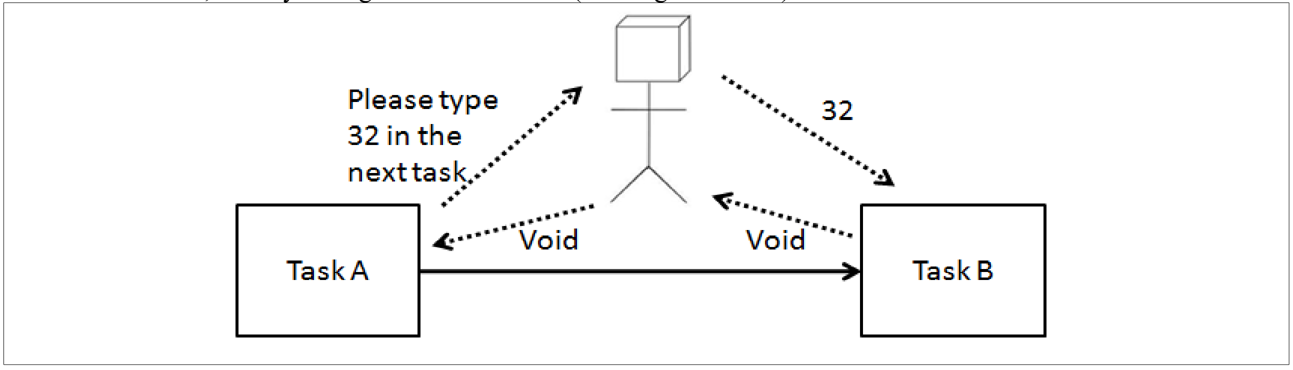
This paragraph explains why some decisions have been taken and what alternatives have been considered.

Activities have been chosen to be as context independent as possible. This means that some entities that are involved in an activity can only deal with the input data they get. Even in the lane example, in which every lane got their own database, this database was designed to be read only and thus always to return the same result given a certain input. The main idea behind this approach is that whenever an activity is fed with the same input, the same output can be expected to a certain extend. In reality this is not completely true, which can already be seen when a user is prompted: The first time he will give a certain answer to a question, the second time perhaps the same answer, but after several times he might get annoyed or fed up and give a different answer to bug the system. However, in case the activity is virtually context free the following advantages arise, which are also advantages of SOA[46]:

- Flexibility: When activities are used in a loosely coupled environment, they can be rearranged in many different ways, since their behavior is predictable.
- Interoperability: When activities follow certain standards they can be used in many different systems under many different conditions.
- Re-usability: Activities that perform deterministic behavior can be reused for solving different problems.

The consequence of sticking to the just mentioned paradigm, requires certain properties have to be obeyed:

- No central database: One could decide to save all the outputs of activities that have taken place before and provide this big collection to the next activity that needs to run sequence wise. In principle nothing is wrong with this approach, since in the proposed data layer from this chapter, inputs of activities can always be mapped to outputs of activities that happened before. However, this gives an activity much more information then it needs and makes it ability get reused much smaller due to a very context dependent signature.
- Limited resources: The way a task is defined in the Itasks system, is as an entity that can have access to anything (the so called *World variable). This includes any kind of I/O (hard drive, user, printer). In the implementation of an atomic task, the choice has been made to limit the resources as much as possible, unless really impossible (such as user prompt) and to let activities have no way to communicate with each other, except for the data flow. The last restriction is not completely the case though. Counter example: Imagine a sequence of task A and task B. Task A asks the user to enter the number 32 in the next task and is of type (Task Void). Task B has no input, but is of type (Task Int). Of course information can be transferred between two tasks now, namely through the human brain (see diagram below).



5.8 Conclusion

This paragraphs has discussed an extension to the BPMN meta model as provided in the previous chapter. This extension involved four steps, namely data layer, activities, triggers and resources. The data layer is defined to describe the flow of data between activities, besides the sequence flow that is already part of the BPMN diagram. For activities, different ways of implementing them have been described and when one wants to make an implementation of a BPMN diagram, one needs to decide one of them for every single activity. For triggers different suggestions have been given, depending on the kind of trigger that needs to be implemented. Finally resources are discussed with respect to lanes and pools. Different lanes in a BPMN diagram, mean that the activities that part of different lanes have access to different resources. While different pools indicate that one is dealing with a completely different system, in which messaging is the only way of communicating. The next chapter will use the population of the extended BPMN meta model (the model introduced in the previous chapter plus the one of this chapter), to generate a Clean implementation. This mapping will be done by hand, but in a very automated way. This makes it assumable that this step can be taken using an algorithm in the future.

6 Mapping the extended BPMN meta model to Itasks

So far the meta models for respectively describing and implementing BPMN in general has been given, also referred to as the extended meta model. This chapter will make the implementation more concrete by linking it to runnable software. In order to implement BPMN in a functional way, Itasks will be used. Itasks will be introduced briefly in the first paragraph, after which one paragraph is dedicated to describing its semantics (or at least the ones that are relevant for the BPMN implementation). After this several paragraphs will deal with mapping all the mentioned BPMN constructs to the Itasks language. This step implicitly leads to operational semantics for BPMN. Pseudo code to achieve this transformation is given as well as the concrete consequences for small examples are shown. This makes it possible to describe the semantics of the BPMN constructs, since they can be decomposed to Itasks building blocks, of which the semantics are known.

6.1 An introduction to Clean and the Itasks system

Appendix B shows the infrastructure of the Itasks system. The main building bricks that are involved here are Clean and the Itasks libraries.

Clean is a functional language (similar to Haskell), that has been developed at the Radboud University in Nijmegen[12][R2][R8]. Some properties that make the language interesting for scientific purposes are:

- Lazy evaluation: Things are only evaluated when truly necessary. This makes it possible to deal with practically impossible constructs such as infinite lists, as long as they do not need to be evaluated completely. Also this causes things that turn out to not be needed in the end, not be evaluated and thus saves valuable computing time. The price that has to be paid for lazy evaluation, is doing more administration and exactly knowing what you are doing.
- Strongly typed: All the types of things that are connected need to fit data type wise.
- Purely functional: Everything programmed is written as a function (opposed to the Object Oriented or imperative paradigm)
- Functions without side effects: A function will always behave the same, independent of context. This can be done by allowing a function to only deal with its input parameters and nothing else.
- Higher order: Functions are first class citizens, which means functions can be parameters and results of functions.
- I/O support: Unlike some other functional languages, Clean does support I/O operations, which are an indispensable part of a workflow system.
- Efficient running time of compiled code: Clean scores high on benchmarks in comparison to other languages such as Haskell and OCaml[R38][R39].
- Easy optimization of code.

Especially the first four properties are interesting in the BPMN context, since:

- One does not need to figure out what needs to be done next
- One is sure that everything fits before run time
- Components can be reused because they will always behave the same
- Activities that occur in BPMN are very similar to functions.

In addition to this the higher order property makes it possible to have workflows and tasks that reason about workflows and/or tasks. For example, one task might encompass the creation of a workflow. This is especially interesting in the context of management of a system, because it allows reflection.

The just mentioned information about Clean, has driven the researchers in Nijmegen to develop a system called Itasks[9][R3]. In short this is a workflow management suite written in the Clean language. The definition of workflows happens by writing Clean code, but instead of having to build workflows from the scratch, the Itask library offers constructs to do so. Very generic business constructs as proposed by Van der Aalst[13] are part of the library and by combining them one can virtually make and workflow as wished. In addition to this all, the Clean constructs such as recursion, higher order functions, lazyness and generics can be used in combination with this. However, having too much expressiveness and too many options can lead to problems. The weaknesses of having too many possibilities are as follows:

- It is hard for business people to understand what is going on (concepts such as recursion might be unknown or hard to imagine to them).
- It is hard to represent graphically what is happening, even though this often has turned out to be the most

convenient way for humans to communicate.
- Things can be solved in many different ways all leading to the same result, causing confusion how to deal with certain problems.
- Business people can not or are not assumed to read and write Clean code.

In order to overcome these problems, this thesis has used the BPMN notation as its starting point. The recognition of BPMN by the business community as result, causes the Itasks system to be more accessible to the business community this way. The price that needs to be paid for this however, is that some functionality is sacrificed, no matter how elegant or useful.

6.2 Describing the semantics of Itasks

Appendix D offers an overview of a subset (namely the ones relevant to BPMN) of the building blocks of Itasks, of which the semantics are explained in a scientific paper[11]. The building blocks are used in the following paragraphs to describe the BPMN concepts as mentioned in this paper. In addition to this some standard Clean concepts are used of which the semantics are either straightforward or explained briefly when needed.

All the building blocks offer a way to create a task, which is basically a description of bunch of work that needs to be finished sometime once an initialization takes place. The infrastructure as mentioned in the previous paragraph offers the startEngine function to plug tasks into the system. In addition to this Start is the main function that is evaluated when a Clean program is started:

```
Start :: *World -> *World
Start world = startEngine workflows world
where
 workflows =
 [
  { name = "Task1", label = "Task number 1" , roles = [], mainTask = task1},
  { name = "Task2", label = "Task number 2" , roles = [], mainTask = task2},
  { name = "Task3", label = "Task number 3" , roles = [], mainTask = task3}]
```

When the above Clean code is compiled and ran, the Itask engine will be started with three task descriptions, namely task1, task2 and task3. Through the browser it is then possible to access the Itask system and to make as many instances as one likes of the different tasks.

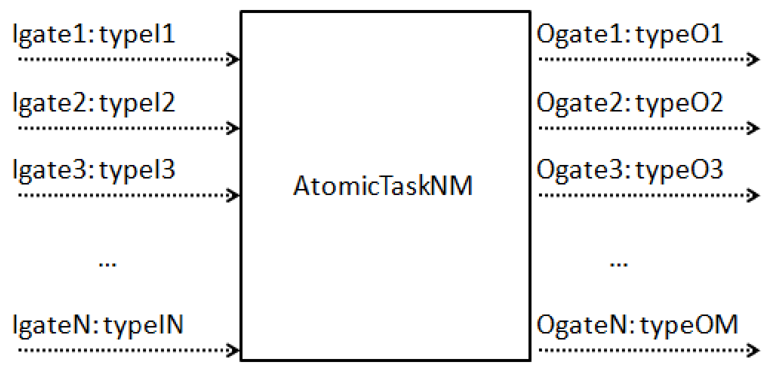
An example of a simple task, which requires and integer to be entered, is as follows:

```
EnterIntegerTask :: Task Int
EnterIntegerTask = enterInformation
```

Appendix F shows that a task can be far more then a simple piece of work that leads to some information. This is illustrated using the Google maps engine with several interactions taking place.

6.3 Implementing atomic activities in Itasks

For the atomic activities in BPMN five approaches have been introduced before, namely user interaction, abstraction, calculation, external interaction and data access. First some general start up work is done which is relevant for each of the implementations, after which the specific implementation for each of them is given. Let the atomic activity be of the following form:



The above means that the task has N input and M output gates, which are labeled Igate1 to IgateN and Ogate1 to OgateM respectively. Their types are typeI1 to typeIN and typeO1 to typeOM. Given this generic activity (which stands for all the atomic tasks that are possible) an implementation for all five cases as given before needs to be given, which is done later on in this paragraph after the generic starting approach.

Since functions can only return one value, something smart needs to happen in order to support M values to be returned. This is achieved by returning a record of size M. For cosmetic reasons and re-usability reasons, the input is also represented as a record (of size N). This gives the following code in Clean for defining the input and output data type:

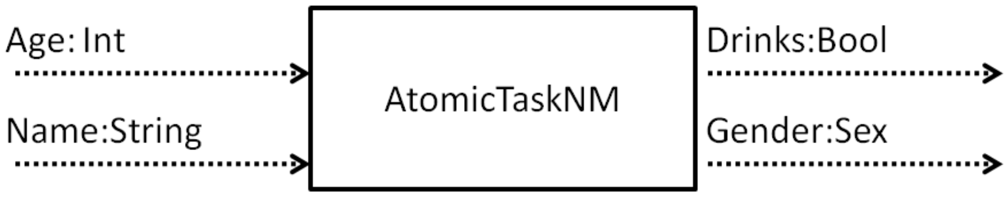
```
:: AtomicTaskNMInput =
 { igate1 :: TypeI1
 , igate2 :: TypeI2
 , igate3 :: TypeI3
...
 , igateN :: TypeIN
 }

:: AtomicTaskNMOutput =
 { ogate1 :: TypeO1
 , ogate2 :: TypeO2
 , ogate3 :: TypeO3
...
 , ogateM :: TypeOM
 }
```

With ... replaced with all the input and output gates in between the ones shown.

For the concrete examples, the following specific data types are used:

```
:: Sex = Male | Female
:: AtomicTaskNMInput =
 { age     :: Int
 , name    :: String }
:: AtomicTaskNMOutput =
 { drinks  :: Bool
 , gender  :: Sex }
```

This corresponds to the following specific diagram for the atomic task:



After this generic introduction the five implementation approaches differ.

6.3.1 Implementing the user interaction in Itasks

For the 'User interaction' variant the resulting code becomes:

```
AtomicTaskNM :: UserId AtomicTaskNMInput -> (Task AtomicTaskNMOutput)
AtomicTaskNM user input = user @:("Perform task atomicTaskNM please",
 (showStickyMessageAbout "" input) ||- (enterInformation ""))
```

The task expects two inputs, namely the input record structure as defined before and the user whom should perform the task. The input is shown to the user, after which the user can finish the task by entering information for all the output gates.

The reason why the user is included in this approach, becomes more clear later when the concept of lanes in BPMN is implemented. No assumptions can be made about how the input gates and output gates are related, even when their types match in some sort of way and therefore the approach as just given is the best one can do. When a link between input and output is known, different options are possible, but only after the meta model has been extended to support this or matches between gates can be determined otherwise (eg by their order or their name). In that case an edit task becomes an option, instead of having to create all the output from the scratch.

Appendix F shows that the generation (or entering) of information by the user can be more then just a one way process from the user to the system. Due to the use of many interactions and the functionality Google Maps offers, an advanced way to generate information becomes possible.

The following code gives a complete (no more code is needed then shown below), concrete example of a user interaction:

```
module atomicTask

import iTasks

// Step 1: Define all the types needed
:: Sex = Male | Female
:: AtomicTaskNMInput =
 { age     :: Int
 , name    :: String
 }
:: AtomicTaskNMOutput =
 { drinks  :: Bool
 , gender  :: Sex
 }

// Step 2: Make derivations for all the types
derive class iTask AtomicTaskNMInput, AtomicTaskNMOutput, Sex
derive bimap (,), Maybe

// Step 3: Define the task
AtomicTaskNM :: UserId AtomicTaskNMInput -> (Task AtomicTaskNMOutput)
AtomicTaskNM user input =
 user @:("Perform task atomicTaskNM please",
 showStickyMessageAbout "" input ||- (enterInformation ""))

// Step 4: Init the workflow engine with the single task created
Start :: *World -> *World
Start world = startEngine workflowitems world
where
      workflowitems = [ workflow "SpecificAtomicTaskNM"
                          (AtomicTaskNM "root" {age=6, name="Peter"})
```

The code creates a workflow engine server with one workflow. The workflow encompasses the atomicTask which is implementated as a user prompt. The input of the automic task are an age and name, and the user performing the task (in this case root) is supposed to enter whether this person drinks and what the gender of this person is. The pseudo code for the algorithm to generate the above code from the meta population (and of which the steps are annotated in the code) is:

```
1) Define all the datatypes needed for the atomic tasks. This includes all the input and output, but also definitions that are use in the input and output
2) Derive gPrint, gParse, gVisualize an gUpdate (class iTask) for all the datatypes defined in step 1, in order to make it possible for the Itasks engine to deal with them
3) Define the task as a combination of showing input to the user and prompting for output
4) Start the workflow engine with the task as defined in step 3
```

For the examples after the just shown one in the next subparagraphs only step 3 will be shown, since the other steps are basically the same for each kind of implementation.

6.3.2 Implementing an abstraction in Itasks

For the 'Abstraction' option, the assumption is that there is already a business process at hand which can perform the task. The input gates are linked correctly to the start event and the output gates to the stop event. The code is therefore very straightforward:

```
AtomicTaskNM :: UserId AtomicTaskNMInput -> (Task AtomicTaskNMOutput)
AtomicTaskNM user input = BusinessProcessNM user input

BusinessProcessNM :: UserId AtomicTaskNMInput -> (Task AtomicTaskNMOutput)
// The code of the business process goes here
```

When currying is used, user and input can even be omitted from both sides of the = sign. The signature of the business process used is the same as the one for the encompassing atomic task. This means that every business process can be treated as an atomic task, without doing extra any effort.

6.3.3 Implementing a calculation in Itasks

The 'Calculation' approach only uses Clean code to do manipulations and can only have access to the input variables. This means for example that I/O is forbidden (reading from a disc, asking a user something), since the calculation is supposed to always return the same result given the same input. The 'return' function can be used to lift the result of the mathematical calculation to something that is considered a task. The code becomes:

```
AtomicTaskNM :: UserId AtomicTaskNMInput -> (Task AtomicTaskNMOutput)
AtomicTaskNM _ input = return (MathematicalCalculation input)

MathematicalCalculation :: AtomicTaskNMInput -> AtomicTaskNMOutput
// The code for implementing MathematicalCalculation goes here
```

An example for the mathematical function which gives all the prime numbers up to a certain number (which has been implemented using three extra helper functions) is as follows:

```
:: AtomicTaskNMInput =   { maxPrime  :: Int   }
:: AtomicTaskNMOutput =  { primeList :: [Int] }

MathematicalCalculation :: AtomicTaskNMInput -> AtomicTaskNMOutput
MathematicalCalculation input =
   { primeList = (GivePrimesFromTo 2 input.maxPrime) }

GivePrimesFromTo :: Int Int -> [Int]
GivePrimesFromTo fr to
| fr > to            = []
| isPrime fr         = [fr:theRest]
| otherwise          = theRest
where
    theRest = GivePrimesFromTo (fr+1) to

isPrime :: Int -> Bool
isPrime number =  allFalse[(number/div)*div == number \\ div <- [2..number-1]]

allFalse :: [Bool] -> Bool
allFalse ax = length (filter (\x -> x==True) ax) == 0
```

6.3.4 Implementing an external interaction in Itasks

The 'External interaction' approach is rather complicated, because no assumptions can be made about the web service since it is an external entity. Since web services only work with data type String, two conversion functions are needed, namely a printing function and parsing function.

Appendix E shows the addition that has been made to the Itasks infrastructure in order to make RPC calls possible. This infrastructure is used in a Google Latitude example, which illustrates what truly needs to be done in order to make a RPC. The following steps are taken:

- Selecting a web service which will be included in Itasks as a atomic activity
- Investigating the needed data types
- Generate a IDL file for the RPC call
- Feed the IDL file to the stub generator and embed the resulting stub in the Itasks code

Most of the steps above are done manually, but all are automatable. First the selection of the web service. The Google Latitude web service[R10], which looks up a location of a person somewhere on the world in near real time has been chosen, which can be accessed through a URL[R11]. It takes two parameters, namely a type and a user. For type "json" will be chosen (in String format), since Clean automatically comes with a JSON parser. For the user field, some very long number (which will be represented as a String) is needed. Since not everyone who uses Google Latitude wants their location to be shared publicly, most of the long numbers are unused and assigned somewhat at random to new users all the time. By searching for the URL itself on Google[R12] as mentioned in the practicum manual for a BPM course[R13], it becomes possible to obtain hundreds of user ids which can be used for testing purposes.

The URL
http://www.google.com/latitude/apps/badge/api?user=-7654330784057799570&type=json
is a concrete initialization of a Google Latitude location request, and results at the moment of writing in the following response (indicating the location of, Freek van den Berg, the person who wrote this document):

```
{ "type": "FeatureCollection",
  "features": [ { "type"    : "Feature",
                  "geometry":{"type": "Point",
                              "coordinates": [5.86879, 51.824302]},
                              "properties": { "id": "-7654330784057799570",
                                              "accuracyInMeters": 847,
                                              "timeStamp": 1265111312,
                                              "reverseGeocode": "Nijmegen, The
Netherlands",
                                              "photoUrl":
"http://www.google.com/latitude/apps/badge/api?
type=photo&photo=TEcnFCoBAAA.HlW7Pec96Gz4s0TBB9eO2Q.fEkEpa507YLVffUvkpCnFw",
                                              "photoWidth": 96,
                                              "photoHeight": 96,
                                              "placardUrl":
"http://www.google.com/latitude/apps/badge/api?
type=photo_placard&photo=TEcnFCoBAAA.HlW7Pec96Gz4s0TBB9eO2Q.fEkEpa507YLVffUvkpCnFw&movin
g=false&stale=true&lod=1&format=png",
                                              "placardWidth": 56,
                                              "placardHeight": 59
                              }
               }
            ]
}
```

The URLs have been aligned to the left, in order to make them fit better into this document. The above structure looks liked a nested record structure. In order to use the stub generator, the following IDL file needs to be generated, taking the type and user as input and returning the above result in FeatureCollection format (which later on can easily be parsed using the JSON parser when the type definition for FeatureCollection is defined):

```
{ "service" :
      { "name"        : "Google Latitude Service"
      , "description" : "A Way to localize people on the world"
      }
, "interface" :
      { "protocol" : ["HTTP","GET"]
      , "type"     : "JSONRPC"
      }
, "operations" : [
      { "name" : "Localize person"
      , "parameters" : [
            { "name" : "user"
            , "type" : "RPCString"
            },
            { "name" : "type"
            , "type" : "RPCString"
            }]
      , "location"   : "http://www.google.com/latitude/apps/badge/api"
      , "callType"   : "RequestResponse"
      , "returnType" : "String"
      }]
}
```

Feeding the IDL to the stubgenerator, gives the following code for the RPC:

```
implementation module google_latitude_service

import JSON
import RPC
import TSt

localize_person :: String String (String -> String) -> Task String
localize_person user type parsefun = mkRpcTask "Localize person"
       { RPCExecute
       | taskId = ""
       , interface = { RPCInterface | protocol = HTTP GET, type = JSONRPC }
       , operation = { RPCOperation | name = "Localize person", parameters = [{ RPCParam
| name = "user", type = RPCString }, { RPCParam | name = "type", type = RPCString }],
location = "\
http://www.google.com/latitude/apps/badge/api", callType = RequestResponse, returnType =
"String" }
       , paramValues = [{ RPCParamValue | name="user", serializedValue = toString user},
{ RPCParamValue | name="type", serializedValue = toString type}]
       , status = ""
       } parsefun
```

A parser from String to String is needed, which deals with the result of the call. Since the JAVA daemon returns the result in base64, base64Decode is used later on the obtain the result actually meant.

Next this stub needs to be implemented in the standard code for an atomic activity. This leads to the following code, which can be generated in a way similar to the previously mentioned activities:

```
module latitude

import iTasks
import JSON
import RPC
import TSt
import Base64

:: AtomicTaskNMInput  = { googleID :: String }
:: AtomicTaskNMOutput = { googleLocation :: String }

derive class iTask AtomicTaskNMInput, AtomicTaskNMOutput
derive bimap (,), Maybe

AtomicTaskNM :: UserId AtomicTaskNMInput -> (Task AtomicTaskNMOutput)
AtomicTaskNM user input =
   user @:("Perform task atomicTaskNM please",
   localize_person "json" input.googleID base64Decode
   >>= \output -> return { googleLocation = output} )

localize_person :: String String (String -> String) -> Task String
localize_person user type parsefun = mkRpcTask "Localize person"
       { RPCExecute
       | taskId = ""
       , interface = { RPCInterface | protocol = HTTP GET, type = JSONRPC }
       , operation = { RPCOperation | name = "Localize person", parameters = [{ RPCParam
| name = "user", type = RPCString }, { RPCParam | name = "type", type = RPCString }],
location = "\http://www.google.com/latitude/apps/badge/api", callType = RequestResponse,
returnType = "String" }
       , paramValues = [{ RPCParamValue | name="user", serializedValue = toString user},
{ RPCParamValue | name="type", serializedValue = toString type}]
       , status = ""
       } parsefun

Start :: *World -> *World
Start world = startEngine workflowitems world
where
       workflowitems = [ workflow "SpecificAtomicTaskNM"
                         (AtomicTaskNM "root" {googleID="-7654330784057799570"}
                         >>= \output -> showMessageAbout "" output) ]
```

Additionally data types can be defined, which are described in the GeoJSON specification[R14]. This specification als contains FeatureCollection which the RPC call returns. When this is done rightly, one can decide to make a task of type FeatureCollection instead of String and have everything directly parsed by using the generic fromJSON function which is part of the Clean library. Also it is possible to take a subset of the data type, using the Clean functions for JSON of which JSONtree is one. These steps are not taken in this thesis.

Once the data type has been derived correctly either way, it can be converted to type GoogleMap, which makes it possible to print the location, as is shown in Appendix F in the second example.

6.3.5 Implementing data access in Itasks

Finally the 'Data access' approach, which looks up the requested output in a database using the information of the input gates as a key. The database therefore stores one table (a list) with tuples of data type (atomicTaskNMInput, atomicTaskNMOutput). As mentioned before there is a different database for every lane in the BPMN diagram. This in order to simulate that there are resources that can not be accessed by everyone. In the just shown examples the definition of a lane has been simulated by including something of data type 'User' in all the functions. The next example will continue using this approach. The DBOfUser function converts a user to something in String format, which is used to obtain a unique database per user. This requires DBOfUser to be bijective.

```
:: DBtuple :== ( AtomicTaskNMInput, AtomicTaskNMOutput)
:: DBtuples :== [DBtuple]

//Converts a UserId
DBOfUser:: UserId -> String
DBOfUser user = toString user

UserDBId:: UserId -> DBid DBtuples
UserDBId user = mkDBid (DBOfUser user) LSTxtFile

//Read the database and returns its contents in task form
ReadDBofUser:: UserId -> Task DBtuples
ReadDBofUser user = readDB (UserDBId user)

//Looks up a record in a database and returns its output value
FindInDB:: DBtuples AtomicTaskNMInput -> AtomicTaskNMOutput
FindInDB [] _ = Abort "Record not found at all"
FindInDB [(i,o):iox] input
| i == input = o                     // Match. Return the value.
| otherwise = FindInDB iox input     // No match. Search in the rest of the
                                     // database

AtomicTaskNM:: UserId AtomicTaskNMInput -> (Task AtomicTaskNMOutput)
AtomicTaskNM user input =
  ReadDBofUser user
  >>= \DBcontents -> FindInDB DBcontents input
  >>= \DBmatch -> return DBmatch
```

The database functions mkDBid and readDB are used from the Clean libraries and are also part of the Itasks library by default. The program crashes when the requested key is not part of the database. Another option would be to return a default value. When a key occurs more then once in the database, the first one is taken. Referring back to the first example a concrete database could look as follows:

Database	
Input	**Output**
`{ age=15, name="Alice" }`	`{ drinks=True,  sex=Female }`
`{ age=25, name="Bob" }`	`{ drinks=False, sex=Male }`
`{ age=17, name="Carl" }`	`{ drinks=False, sex=Male }`
`{ age=19, name="Derek" }`	`{ drinks=True,  sex=Male }`
`{ age=34, name="Ernie" }`	`{ drinks=False, sex=Male }`
`{ age=56, name="Francesca" }`	`{ drinks=True,  sex=Female }`
`{ age=16, name=”Gary” }`	`{ drinks=False, sex=Male }`

6.3.6 Defining a generic atomic activity in Itasks

All the code given so far can still be improved in certain ways, which however will not be done in this thesis in order to not loose focus:

- Instead of mentioning the input and output gates in a type, the function could be made polymorphic with respect to them. This makes it possible to reuse the code for atomic activities for different activities to a big extend. This avoids a lot of duplicate, redundant and possibly inconsistent (when partly updated) code.
- All the five implementations could be merged together in a new kind of atomic activity. This requires the new kind to have an extra parameter to indicate which implementation is supposed to be used. The code below shows a brief example of this.

Code for an implementation independent atomic activity (in which the ... can be replaced by the implementations as given before):

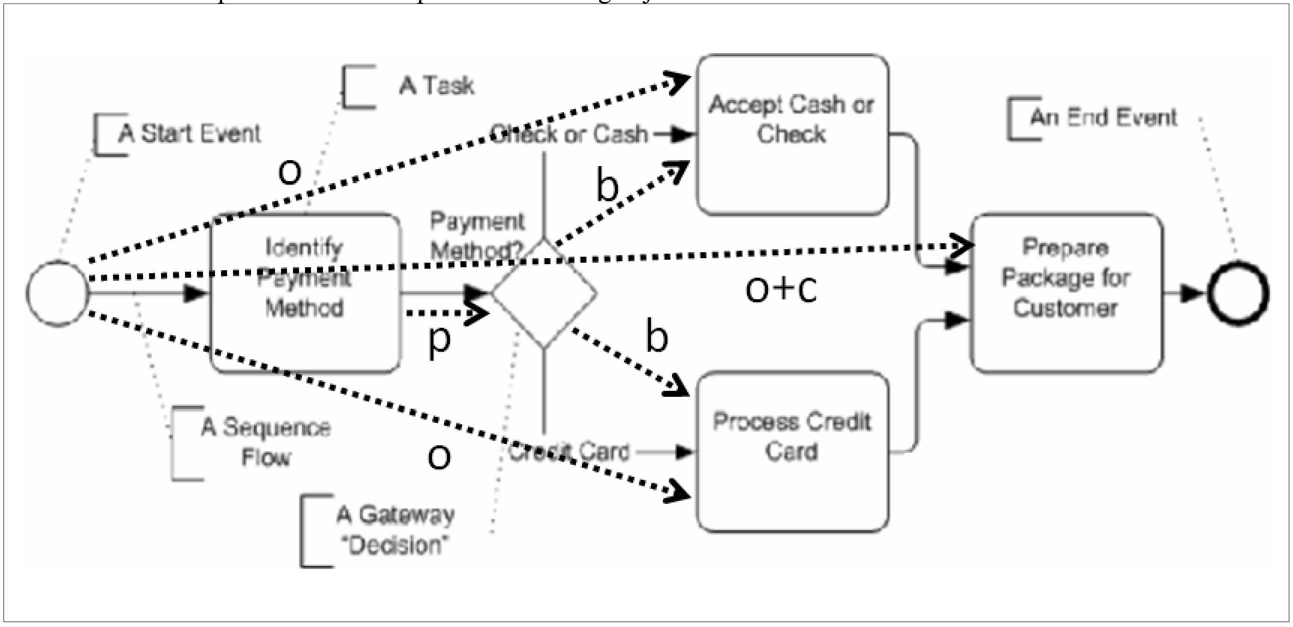
```
:: AtomicTaskImplementationKind =
       ItasksUserPrompt
       | DecomposedBusinessProcess
                       (UserId AtomicTaskNMInput -> (Task AtomicTaskNMOutput))
       | MathematicalCalculation (AtomicTaskNMInput -> AtomicTaskNMOutput)
       | WebService (AtomicTaskNMInput -> Task AtomicTaskNMOutput)
       | DataSource (UserId -> String)

AtomicTaskNM :: AtomicTaskImplementationKind UserId AtomicTaskNMInput
                                                 -> (Task AtomicTaskNMOutput)
AtomicTaskNM UserInteraction user input                        = ...
AtomicTaskNM (Abstraction bp) user input                       = ...
AtomicTaskNM (Calculation mc) user input                       = ...
AtomicTaskNM (ExternalInteraction stub) user input             = ...
AtomicTaskNM (DataAccess ds ) user input                       = ...
```

In the above code the type 'AtomicTaskImplementationKind' has been introduced, but will not be used later on. For the five options the explanation of parameters are as follows:

Option	**Explanation**
UserInteraction	Does not need any extra parameters, because the showMessage and enterInformation task know what to do just by having an input and output type at hand.
Abstraction	The actual process that is going to implement this atomic activity is given as a parameter
Calculation	A Clean function that transforms input to output (without caring about tasks) is given as parameter.
ExternalInteraction	The stub that has been given before is accessed, but wrapped in something that matches the atomic activity layout as mentioned so far. It can be fed with the input parameter, which is already part of the atomic activity.
DataAccess	An extra function that assigns a unique data source to every user is added. Since the UserId was already part of the implementation, the input of this function is already at hand.

6.4 *Implementing the sequential and data flow in Itasks*

In order to implement multiple activities that are connected in a BPMN diagram the sequential and data flow needs to be considered. Recap the BPMN example for connecting objects:



The following abbreviations have been used:

- O: Order data
- C: Customer data
- B: Boolean
- P: Payment method

For the atomic activities the following implementation has been chosen (which might not be completely realistic, but very convenient for testing):

Activity	**Implementation**
Identify payment method	Ask the user how he wishes to pay
Accept Cash or Check	Ask the user to press okay when the customer has paid, given the order data (to obtain the price).
Process Credit Card	Ask the user to enter the credit card information, given the order data (to obtain the price).
Prepare package for customer	Ask the user to press okay when the package is packed, given the order data (to obtain what to sent) and customer data (to obtain to whom to address it).

Since pools and lanes are not part of the deal yet, the user is chosen to be constant in the above implementation. This could be for example a person working at the company, which observes what happens in the real world and presses buttons when certain events occur (Eg when a payment is made). In addition to this multiple people can operate the system using one single account, which in many cases might get rid of the need to have multiple accounts in the system. When one only cares about the data that flows around, it might seem that the sequential flow has no meaning at all. This is however not the case, since atomic activities can have side effects (Eg writing to a database, sending an SMS), which can not been seen by just looking at the data flow. It is therefore important to let the sequential flow be the leading one, since otherwise the order of execution can influence the final result of the business process and side effects might depend on each other. In principle there is a clear linear order in which activities are executed, but when a gateway is part of the diagram, it can not be predicted what way to go.

The above diagram leads to the following Clean code in which every statement in the code is either fixed or from the meta model population as defined before:

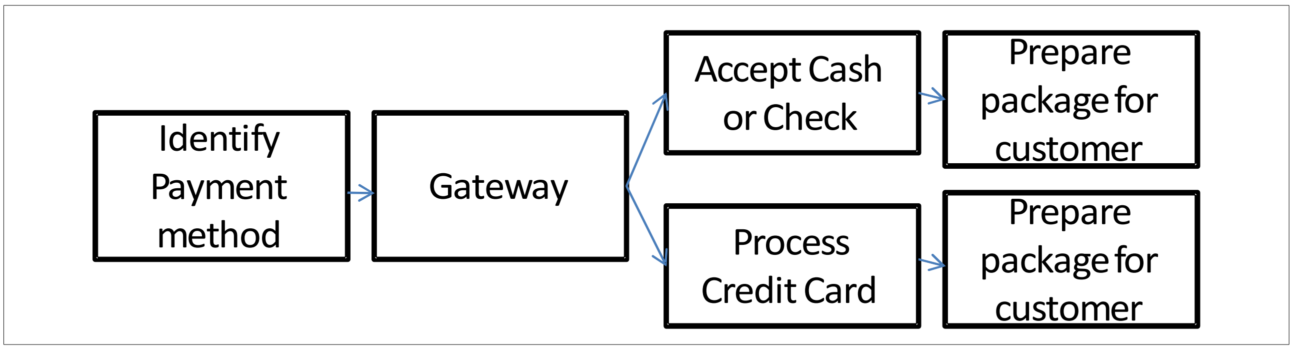
```
module sequentialFlow

import iTasks

//Step 1: Used data types
:: OrderData = { products :: [Product],
                 price :: Int }
:: Product :== String
:: CustomerData = { name :: String,
                    address :: String }
:: PaymentMethod = CreditCard | Cash | Check

//Step 2: Derive datatypes from step 1
derive class iTask OrderData, CustomerData, PaymentMethod
derive bimap (,), Maybe

//Step 3: Inputs and outputs of atomic activities that are not Void
:: SequentialFlowInput =
              { orderData :: OrderData,
                customerData :: CustomerData }
:: IdentifyPaymentMethodOutput = { paymentMethod :: PaymentMethod }
:: AcceptCashOrCheckInput = { orderData3 :: OrderData }
:: ProcessCreditCardInput = { orderData4 :: OrderData }
:: PreparePackageForCustomerInput =
              { orderData2 :: OrderData,
                customerData2 :: CustomerData }

//Step 4: Derive datatypes from step 3
derive class iTask SequentialFlowInput, IdentifyPaymentMethodOutput,
                   AcceptCashOrCheckInput, ProcessCreditCardInput,
                   PreparePackageForCustomerInput

//Step 5: Define constants for start event data
newOrder :: OrderData
newOrder = { products = ["Hammer","Nails","Screwdriver"] , price = 2995 }

newCustomer :: CustomerData
newCustomer = { name = "John Johnson" , address = "112 Holland Road" }

//Step 6: Define atomic tasks
IdentifyPaymentMethodTask :: UserId Void -> (Task IdentifyPaymentMethodOutput)
IdentifyPaymentMethodTask user input =
 user @:("Perform task IdentifyPaymentMethodTask please",
 showStickyMessageAbout "" input ||- (enterInformation ""))

AcceptCashOrCheckTask :: UserId AcceptCashOrCheckInput -> (Task Void)
AcceptCashOrCheckTask user input =
 user @:("Perform task AcceptCashOrCheckTask please",
 showStickyMessageAbout "" input ||- (enterInformation ""))

ProcessCreditCardTask :: UserId ProcessCreditCardInput -> (Task Void)
ProcessCreditCardTask user input =
 user @:("Perform task ProcessCreditCardTask please",
 showStickyMessageAbout "" input ||- (enterInformation ""))

PreparePackageForCustomerTask :: UserId PreparePackageForCustomerInput
                                                             -> (Task Void)
PreparePackageForCustomerTask user input =
 user @:("Perform task PreparePackageForCustomerTask please",
 showStickyMessageAbout "" input ||- (enterInformation ""))
```

```
//Step 7: Write functionality for every gateway
Eq :: PaymentMethod PaymentMethod -> Bool
Eq Cash Cash               = True
Eq Check Check             = True
Eq CreditCard CreditCard   = True
Eq _ _                     = False

GatewaySelector :: IdentifyPaymentMethodOutput -> Task Bool
GatewaySelector input
| Eq input.paymentMethod Cash           = return True
| Eq input.paymentMethod Check          = return True
| otherwise                             = return False

//Step 8: Translate the activity sequence to one major task, using combinators
SequentialFlowTask :: UserId SequentialFlowInput -> Task Void
SequentialFlowTask user input =
  IdentifyPaymentMethodTask user Void
  >>= \output              -> GatewaySelector output
  >>= \gatewaySelection    ->
      if gatewaySelection
       (
         AcceptCashOrCheckTask user { orderData3 = input.orderData }
         >>| PreparePackageForCustomerTask user
                     { orderData2 = input.orderData,
                       customerData2 = input.customerData }
       )
       (
         ProcessCreditCardTask user { orderData4 = input.orderData }
         >>| PreparePackageForCustomerTask user
                     { orderData2 = input.orderData,
                       customerData2 = input.customerData }
        )



//Step 9: Define the Itasks engine as start event and add the task from
//step 8 as a workflow to it
Start :: *World -> *World
Start world = startEngine workflowitems world
where
  workflowitems =
      [ workflow "SequentialFlowTask"
       (SequentialFlowTask "root"
        { customerData =   newCustomer, orderData = newOrder }
       )
      ]
```

This code that has been generated, is generated using the following algorithm in pseudo code, of which the steps are annotated in the code above:

```
1) Make a list of all the data types used in the BPMN diagram and define these data types in terms of Clean building blocks
2) Derive gPrint, gParse, gVisualize and gUpdate (class iTask) for the types of step 1
3) Make a list of data types for each atomic function's input and output. If the function's input or output is Void, no work is needed here for respectively the input or the output.
4) Derive gPrint, gParse, gVisualize and gUpdate (class iTask) for the types of step 3
5) Write constants for the input of the start event
6) Create tasks for every atomic activity in a way a presented in the previous paragraph. Depending on the kind of implementation, the corresponding block of code needs to be chosen
7) Write an equal function for every gateway and mention every positive case that is part of the population of the meta model. The default case can then be false (all other options).
8) Identify all the possible sequence flows and combine them into one task, using if statements for every gateway selection.
9) Execute the Itasks engine and include the task from step 8 as a workflow
```

For step 8 the simplification as shown down here is used in order to identify flows. The simplification simply does not care that different flows might merge at some later time (such as in the above example happens just before the 'Prepare package from customer'). This might lead to duplicate code in the implementation, but makes the generation of the implementation easier to be automated. The different paths that are available in the BPMN can then be visualized as follows:



Note that in the data type definitions in the code, attributes have been called orderData2, orderData3 etc. in addition to just the regular orderData. This is done to make explicitly clear later on which data type is meant. Clean does offer nicer constructs for dealing with this problem, but because of pragmatic reasons they have not been used.

6.5 Implementing compound activities in Itasks

The implementation of compound activities encompasses two cases that need to be covered, namely the decomposition of an atomic activity in a business process and the Foreach case. The former case has already been covered two paragraphs ago and looks like this:

```
CompoundTaskNM :: UserId CompoundTaskNMInput -> (Task CompoundTaskNMOutput)
CompoundTaskNM user input = BusinessProcessNM user input

BusinessProcessNM :: UserID CompoundTaskNMInput -> (Task CompoundTaskNMOutput)
// The code of the business process goes here
```

BusinessProcessNM then contains the implementation of the whole compound activity. In order to obtain this implementation, the algorithms as described in this chapter can be applied (recursively).

6.5.1 A currency converter example

The latter case, dealing with a Foreach loop, has the extraordinary property that the data types do not match one to one. Instead of this a set is fed to the compound activity, after which the compound activity is executed repeatedly with having every iteration an element of the set as input. Concretely this looks as follows for a simple example:

```
module chinese

import iTasks

// Step 1: Follow step 1 till 9 from the previous paragraph
:: CompoundTaskNMInput  = { Euro :: Int }
:: CompoundTaskNMOutput = { RMB :: Int }

derive class iTask CompoundTaskNMInput, CompoundTaskNMOutput
derive bimap (,), Maybe

newNumbers :: [CompoundTaskNMInput]
newNumbers = map (\value -> {Euro = value}) [34,68,45,65,103,23,67,56,67]

// Step 3: Run a sequence of tasks, equal to iterating through the forEach
// using the following code
CompoundTaskNM :: UserId [CompoundTaskNMInput] -> (Task [CompoundTaskNMOutput])
CompoundTaskNM user inputList  = sequence "" (map mapAction inputList)
where
       mapAction input = BusinessProcessNM user input

// Step 2: Write the contents of the compound activity as an atomic task
BusinessProcessNM :: UserId CompoundTaskNMInput -> (Task CompoundTaskNMOutput)
BusinessProcessNM _ input = return { RMB = input.Euro * EuroToRMB }
```

```
EuroToRMB :: Int
EuroToRMB = 9

Start :: *World -> *World
Start world = startEngine workflowitems world
where
      workflowitems = [ workflow "forEachCompoundTask"
                              (CompoundTaskNM "root" newNumbers >>=
                               \output -> showStickyMessageAbout "" output )
```

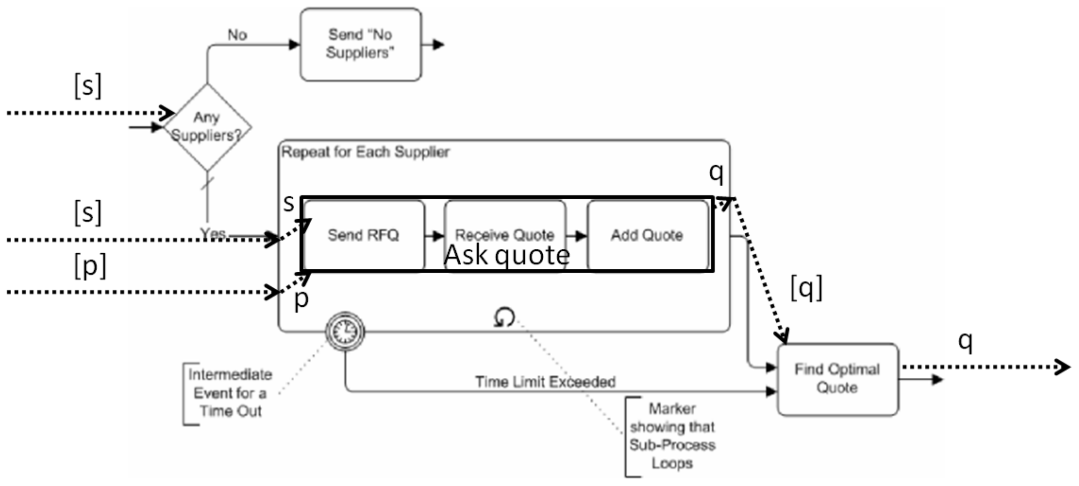
The idea behind this example is that there is a list of amounts in Euros (European money), namely 34,68,45,65,103,23,67,56 and 67. The amounts in this list need to be converted to RMB (Chinese money) amounts, which is done by multiplying them by 9 (at July 23rd, 2010: 1 Euro = 8.76 Chinese yuan), thus resulting in the list 306, 612, 405, 585, 918, 207, 603, 504 and 603. This resulting list is printed to the screen using the showStickyMessageAbout statement for debugging purposes.

The algorithm in pseudo code to generate the code as above (partly annotated in the code) is as follows:

```
1) Follow step 1 to 9 from the pseudo code 'Sequential flow' implementation as given in the previous paragraph.
2) Write the contents of the compound activity as an atomic activity.
3) Use the sequence function of Itasks to run through every iteration of the Foreach loop. Basically this code is fixed and the only thing that needs to be done is inserting the compound activity from step 2.
```

6.5.2 A supplier quoting example

Recap the BPMN example using a Foreach loop (including the addition of fictional data flows), which will be used for the creation of a more advanced example:



The following abbreviations have been used:

- S: Supplier data
- P: Product data
- Q: Quote data (a combination of a supplier, product and price)

The above BPMN diagram does not have a start and stop event. The assumption here is that the diagram is part of a bigger diagram and that thus all the input information needed has been generated at some time and that all the output information needed will be needed sometime. This is represented by data flows coming in from nowhere on the left and data flows going nowhere on the right. The compound activity has been transformed to one atomic activity named 'Ask quote', since in a computer system the three steps are generally one step. Different ways in merging activities will be addressed in this paragraph. The [] a defined as a set of elements in the data type and the insecurity of having a result is expressed by using maybe. The intermediate event is ignored for simplicity reasons now, but will be part of examination in a later paragraph about events.

The resulting code using the pseudo code algorithm just mentioned (combined with the one for sequential flows, since outside this compound activity there is a flow as well) then becomes:

```
module repeatTask

import iTasks

// Basic data types
:: SupplierData = { supplierName :: String, supplierURL :: String }
:: ProductData = { productName :: String }
:: QuoteData = {supplier :: SupplierData, product :: ProductData, price :: Int}
derive class iTask SupplierData, ProductData, QuoteData
derive bimap (,), Maybe

// Data types of inputs and outputs
:: CompoundTaskNMInput    = { supplierData :: SupplierData,
                              productData  :: ProductData }
:: CompoundTaskNMOutput   = { quoteData    :: QuoteData }

:: RepeatTaskInput        :== [CompoundTaskNMInput]
:: RepeatTaskOutput       :== Maybe CompoundTaskNMOutput

:: FindOptimalQuoteInput  :== [CompoundTaskNMOutput]
:: FindOptimalQuoteOutput :== CompoundTaskNMOutput

derive class iTask CompoundTaskNMInput, CompoundTaskNMOutput

// Input of the start event
newSuppliers :: [SupplierData]
newSuppliers = [ { supplierName = sn, supplierURL = su} \\ sn <- names & su <- urls ]
where
       names = ["Alice","Bob","Carl","David"]
       urls                  =         ["www.alice.com/makequote","www.bom.com/quoteAPI",
                "www.carl.com/askprice","www.david.com/quote"]

newProductData :: ProductData
newProductData = { productName = "Screwdriver" }

newProductDataList :: [ProductData]
newProductDataList = [newProductData \\ x <- newSuppliers ]

newCompoundTaskInput :: [CompoundTaskNMInput]
newCompoundTaskInput   =   [   {   supplierData   =   sd,   productData   =   pd   }
                                    \\ sd <- newSuppliers & pd <- newProductDataList ]

// The actual task to be added to the engine
RepeatTask :: UserId RepeatTaskInput -> (Task RepeatTaskOutput)
RepeatTask user []         = return Nothing
RepeatTask user input      = CompoundTaskNM user input
                             >>= \quotes -> FindOptimalQuote user quotes
                             >>= \quote -> return (Just quote)
// Picking the cheapest quote from a list of quotes
FindOptimalQuote :: UserId FindOptimalQuoteInput -> (Task FindOptimalQuoteOutput)
FindOptimalQuote user input = return (MathematicalCalculation input)

MathematicalCalculation :: FindOptimalQuoteInput -> FindOptimalQuoteOutput
MathematicalCalculation   [a:ax] = cheapest a ax
where
       cheapest a []                              = a
       cheapest a [b:bx]
       | a.quoteData.price < b.quoteData.price   = cheapest a bx
       | otherwise                               = cheapest b bx

// Initiating a Foreach loop, which quotes every supplier
CompoundTaskNM :: UserId [CompoundTaskNMInput] -> (Task [CompoundTaskNMOutput])
CompoundTaskNM user inputList  = sequence "" (map mapAction inputList)
where
       mapAction input = BusinessProcessNM user input
```

```
BusinessProcessNM :: UserId CompoundTaskNMInput -> (Task CompoundTaskNMOutput)
BusinessProcessNM _ input = return (receivedQuote input)

// Task for receiving a single quote
receivedQuote :: CompoundTaskNMInput -> CompoundTaskNMOutput
receivedQuote input = { quoteData =
      { supplier = inputSupplier, product = inputProduct,
        price = quoteStub inputSupplierName }                        }
where
      inputSupplier     = input.supplierData
      inputSupplierName = inputSupplier.supplierName
      inputProduct      = input.productData

// Ideally this is done by calling a web service,
// but for simplicity reasons a stub has been used
quoteStub :: String -> Int
quoteStub "Alice" = 56
quoteStub "Bob"   = 51
quoteStub "Carl"  = 48
quoteStub "David" = 93

Start :: *World -> *World
Start world = startEngine workflowitems world
where
      workflowitems = [ workflow "forEachCompoundTask"
                         (RepeatTask "root" newCompoundTaskInput
                         >>= \quote -> showStickyMessageAbout "" quote ) ]
```

It is considerable to use a parallel execution instead of a sequential one for each of the iterations of the Foreach loop, which will not be done here. Itasks does offer a construction for this however. In cases the work can be done by multiple entities, or in cases there is a big I/O wait, the parallel approach might turn out to be beneficial. One has to keep the some consequences in mind however:

- Every single iteration of the Foreach loop might cause side effects and when iterations are mixed these side effect might have unexpected results (eg printing a text to the screen).
- One might not reach performance advantage by doing things in parallel, because the tasks need to be performed by one entity (eg Itasks user prompt) anyways.
- The high efficiency of the Clean language (eg lazy evaluation) might turn out to be enough to reach the maximum performance possible in sequence already.

As mentioned before the activities 'Send RFQ', 'Receive Quote' and 'Add Quote' have been merged together in a new activity named 'Ask quote'. The disadvantage of this approach is that in the implementation code, the old structure has disappeared. This makes for example reverse engineering impossible or makes it hard to convince people that the code is doing what it is supposed to do. That is why in addition to this approach two other solutions are offered. Both solutions require one of the three tasks to be assigned as being the main task. For both solutions the task 'Receive Quote' has gotten this honor, because this one represents best what the compound activity is doing.

First, the introduction of dummy tasks which will just pass on the information they get as input. In the implementation this looks as follows:

```
SendRFQTask :: UserId CompoundTaskNMInput -> ( Task CompoundTaskNMInput )
SendRFQTask userid input = return input

ReceiveQuoteTask :: UserId CompoundTaskNMInput -> ( Task CompoundTaskNMOutput )
// The implementation as shown before for the compound task goes here

AddQuoteTask :: UserId CompoundTaskNMOutput -> ( Task CompoundTaskNMOutput )
AddQuoteTask userid input = return input

CompoundTaskNM :: UserId CompoundTaskNMInput -> ( Task CompoundTaskNMOutput )
CompoundTaskNM user input = SendRFQTask user input
                            >>= \input2 -> ReceiveQuoteTask user input2
                            >>= \input3 -> AddQuoteTask user input3
```

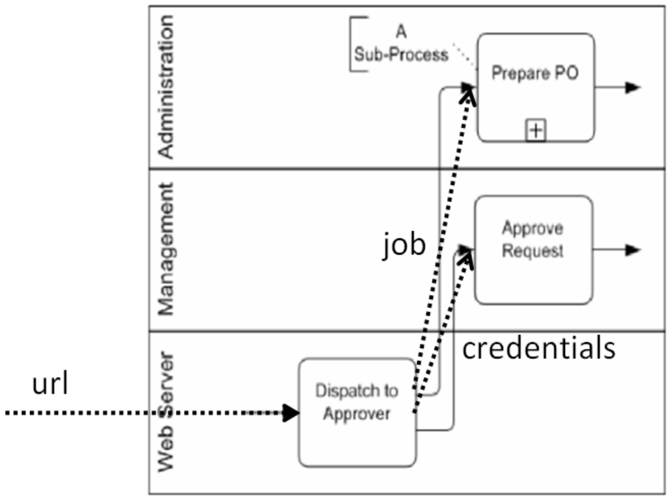
Second, the introduction of empty activities, which are just there for decorative reasons but do not perform any work at all. This is implemented in the following way:

```
SendRFQTask :: UserId Void -> ( Task Void )
SendRFQTask userid input = return Void

ReceiveQuoteTask :: UserId CompoundTaskNMInput -> ( Task CompoundTaskNMOutput )
// The implementation as shown before for the compound task goes here

AddQuoteTask :: UserId Void -> ( Task Void )
AddQuoteTask userid input = return Void

CompoundTaskNM :: UserId CompoundTaskNMInput -> ( Task CompoundTaskNMOutput )
CompoundTaskNM user input = SendRFQTask user Void
                            >>| ReceiveQuoteTask user input
                            >>= \output -> AddQuoteTask user Void
                            >>| return output
```

As can be seen, both implementations respect the sequence order of BPMN and therefore represent better what is going on in BPMN that the initial approach. The reached functionality in both examples is exactly the same as the implementation from before in which all three activities have been merged to a compound one. This means that dealing with either 'ReceiveQuoteTask' or 'CompoundTaskNM' in the two above examples does not make any difference.

6.6 Implementing pools and lanes in Itasks

This paragraph deals with an implementation of pools and lanes. So far the assumption has been that there is only one user and one system to deal with, although activities have been equipped with a user parameter so far anticipating on what was about to come. When pools are introduced, the concept of multiple users comes in place. Fortunately this is supported by the Itasks system without having to do any extra effort, which makes the translation from the meta model quite straightforward. Recall the BPMN diagram about pools as mentioned before to which fictional data flows have been added:



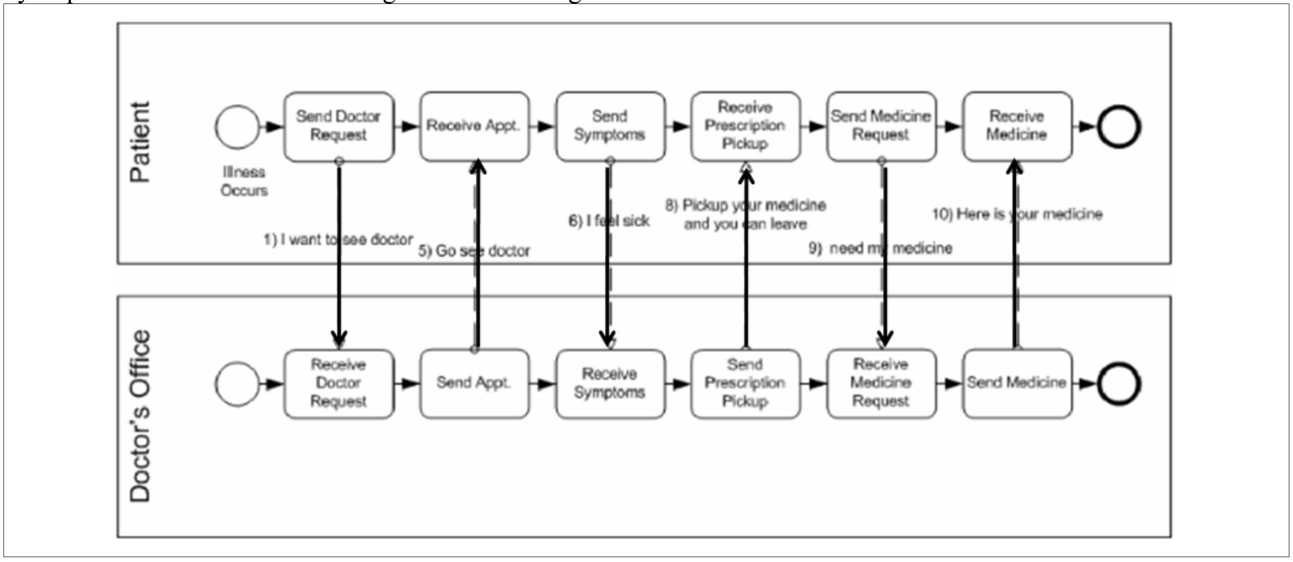
Defining the above data types and then using an algorithm similar to the compound activities and sequences to transform the BPMN diagram to Clean code leads to:

```
module lanes

import iTasks

//Step 1: Used data types
:: URL         = { domain :: String,
                   job :: Job,
                   sessionID :: Credentials }
:: Job         = { relativePath :: String,
                   params :: [String] }
:: Credentials = { username :: String,
                   password :: String }
```

```
//Step 2: Derive data types from step 1
derive class iTask URL, Job, Credentials
derive bimap (,), Maybe

//Step 3: Inputs and outputs of atomic activities that are not Void
:: DispatchToApproverInput = { website :: URL }
:: DispatchToApproverOutput = { job2 :: Job,
                                credentials2 :: Credentials}
:: PreparePOInput = { job3 :: Job }
:: ApproveRequestInput = { credentials3 :: Credentials }
:: LaneTaskInput :== DispatchToApproverInput

//Step 4: Derive datatypes from step 3
derive class iTask DispatchToApproverInput, DispatchToApproverOutput,
                   ApproveRequestInput,PreparePOInput

//Step 5: Define constants for start event data
StartURL :: URL
StartURL = { domain = "clean.ru.nl",
             job = { relativePath = "/callAPI/test", params = ["-runonce"] },
             sessionID = { username = "root", password = "root" }
           }

//Step 6: Define atomic tasks
DispatchToApprover :: UserId DispatchToApproverInput -> (Task DispatchToApproverOutput)
DispatchToApprover _ input = return (MathematicalCalculation input)

MathematicalCalculation :: DispatchToApproverInput -> DispatchToApproverOutput
MathematicalCalculation input =
   { DispatchToApproverOutput | job2 = input.website.job,
                                credentials2 =  input.website.sessionID }

ApproveRequest :: UserId ApproveRequestInput -> (Task Void)
ApproveRequest user input = user @:("Perform task ApproveRequest please",
 (showStickyMessageAbout "" input) ||- (enterInformation ""))

PreparePO :: UserId PreparePOInput -> (Task Void)
PreparePO user input = user @:("Perform task ApproveRequest please",
 (showStickyMessageAbout "" input) ||- (enterInformation ""))

//The actual task to be added to the engine
LaneTask :: UserId LaneTaskInput -> (Task Void)
LaneTask user input = (DispatchToApprover "webserver" input)
       >>= \output -> ApproveRequest "management"
                                    { credentials3=output.credentials2 }
       -&&- PreparePO "admin" { job3=output.job2 }
       >>| return Void

Start :: *World -> *World
Start world = startEngine workflowitems world
where
       workflowitems = [ workflow "LaneTask"
                         (LaneTask "root" {website=StartURL})
```

The key combinator used here is -&&-, which runs two tasks in parallel and merges their results in a tuple. Since both tasks do not produce results, their results are omitted.

When pools are introduced, the situation becomes a bit trickier, because processes are able to exceed the boundaries of one single system. However, if one manages to implement all the pools on one system, message flows could be replaced by sequence flows and the following new BPMN diagram arises:



All message flows have been replaced by sequence flows. One might think that some sequence flows are obsolete now, such as the one between 'Send Doctor Request' and 'Receive Appt', and for the above diagram this is certainly true (since it will eventually be triggered through the 'Receive Doctor Request' and 'Send Appt' path). There are two reasons however why this is not the case in general, which are:

– It never hurts to execute 'Receive Appt' earlier when it does not share side effects with things from the other pool. Therefore keeping the original sequence flow in place as well, prevents blocking when one pool has to wait for another for no reason.
– Imagine if there would be an activity between 'Send Doctor Request' and 'Receive Appt' named 'X', then this activity would not be executed anymore if the sequence between 'Send Doctor Request' and 'X' is removed.

After this step the atomic activities of every pool, but the current system's one, could be replaced by web services, so that the Itasks system is leading and every other system kindly following. This approach offers four downsides:

– One system has to be chosen to be the leading one, which could in real environments lead to delicate cases involving a lot of power and politics.
– The Itasks API does not allow Itasks to be the non-leading system (although this can be achieved by having an automated task that pretends to be an editor and submitter in an enterInformation task, as if it were a human entering it).
– Because of the previous point it is not possible to connect two or more Itasks systems, which seems like something useful.
– The leading system needs to know the business process of all the other systems involved. In reality this is normally not the case and one does not even care how another system solves a problem, as long as the final result is as it wishes. Also this approach makes the non-leading systems very inflexible, because every change in their process has to be communicated to the leading system.

The just mentioned shows that there are reasons to take a different approach. In order to make this approach possible in Itasks, an extension to the API is needed which in Itasks are called handles. It should be possible to leave a message for an atomic activity independent of whether this activity has already been started or not. Since the Clean language does not support multi-threading at the time of writing an extension (in for example the JAVA language as the earlier demonstrated daemon) is needed.

Assume that the just mentioned example with the doctor and patient is implemented using two Itasks systems (since this shows best what the consequences are from both both perspectives). One Itasks system for the patient and one for the the doctor's office. Then informally the following actions take place after the initialization of an instance:

1) An Illness of the patient causes a trigger. An instance of the business process is made at the patient system, since this system is the one being triggered in the business process.
2) The patient makes a map of all pools and systems (including how they can be reached, such as a URL) which from now on will be referred to as system map. This mapping should include not just the system, but also the instance of the process which needs to receive the message.
3) The patient system triggers all systems that take part in the process (in this case only the doctor's office) and gives every starting event the system map as (extra) parameter.
4) Every system runs independently as long as there are no incoming or outgoing message flows

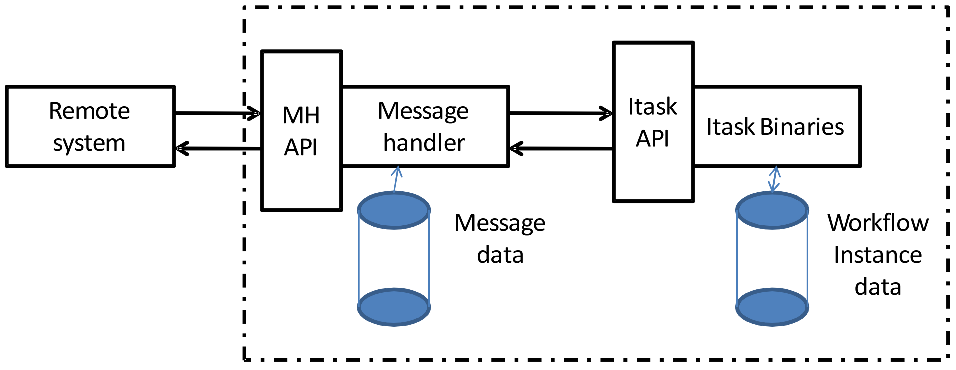
Once a message is being sent or a wait for a message occurs two things can happen, depending on whether respectively the former or latter event occurred. First the sending message case, in which a message is sent while the receiver has not reached the point of receiving it, which results into the following actions to happen:

5a) The sender calls a web service at the system of the receiver with the message it wants to send as a parameter.
6a) The message handler returns a void value as result of the web service so that the sender can continue whatever works it was doing.
7a) The message handler at system of the receiver stores the message, until the receiver is ready to process it
8a) The message handler forwards the message to the receiver when the receiver asks for it (by calling a web service with as parameter the system it wants to receive a message from)

For a receiver reaching the synchronization point first and the message not being sent yet, the outcome is as follows:

5b) The receiver calls the message handler at its current system with a web service call. The input of the call is the system it would like to receive a message from
6b) The message handler does not yet have a message present and just puts the receiver on block by not providing a result yet
7b) The sender is ready to send a message and provides a call to the message handler with the message as input. The message handler returns an acknowledgment to the sender right away.
8b) The message is directly forwarded to the receiver system by the message handler
9b) A void message is by the receiver to the message handler, representing an acknowledgment

What can be concluded from above is that the message handler needs some sort of storage system, which also stores the sender of a message. This is very similar to the email paradigm and therefore the message handler will presumed to have an email box to deal with this. Another point of discussion is whether the sender should immediately continue its job when the message is sent and the receiver has not reached the point of reception yet. Side effects can be a problem in general, but not in this case, since the systems are supposed to be completely independent (except for the messaging part). Given the just mentioned, the global overview of the infrastructure of the message handler and its environment looks like this when applied on the Itasks system (of course every other system is allowed to use any implementation as long as the behavior is as described before):

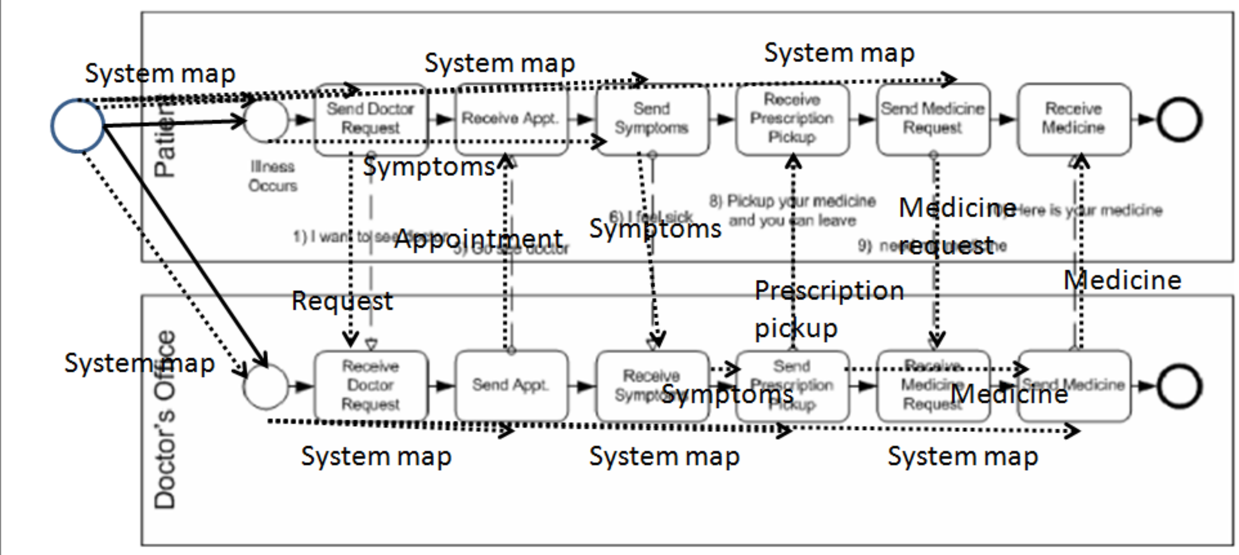


Unfortunately the current Itasks system does not offer this functionality (yet). The message handler system is not available and the Itasks API does not support inserting data into a currently running task instance in a fashionable way. The latter aspect could be achieved by simulating a browser session, in which information is entered right now, but should in a future version of Itasks be implement in a way that makes access by the RPC daemon as shown before possible.

In order to still get an impression and approximation of what is needed for pools and to get a hold on the semantics, some of the above concepts will be done manually during runtime. This is done by giving every activity an extra parameter to represent that input from one or more external systems is needed. This extra parameter for an activity expecting n messages, the previously mentioned system map and new definition of an atomic task result in the following Clean code:

```
:: SystemMap :== [ExternalSystem]

:: ExternalSystem = { name :: String, url :: String }

:: IncomingMessageType =
  { message1 :: Message1Type,
    message2 :: Message2Type,
    message3 :: Message3Type,
    ...
    messageN :: MessageNType
  }

AtomicTaskNM :: IncomingMessageType UserId AtomicTaskNMInput → (Task AtomicTaskNMOutput)
// Implementation of task goes here
```

When the data layer is added to the example BPMN diagram about pools, the following picture appears:



An extra trigger has been added before the patient trigger, which triggers all pools. It is this new trigger that is executed when an illness occurs instead of the original one. The system map is needed in every activity that involves sending a message and whenever a message is sent, a mapping using this system map occurs. With respect to the above diagram, the Doctor's Office's system will be implemented by Itasks, while the patient pool represents an interface to the patient of any kind (eg. through the phone, through visiting the pharmacy or by sending letters).

Due to the inability of Itasks to deal with a message handler at the time of writing, a new Itasks user has been created named 'messagehandler' to deal with the concept. Three functions delegating to this user will simulate the sending and receiving of messages are introduced and are explained later.

The following code implements the business process of the doctor's office and leaves it open to the patient how to implement his process as long as it synchronizes with the doctor's office rightly:

```
module pools

import iTasks

//Step 1: Used data types
:: Month     :== Int
:: Day       :== Int
:: Hour      :== Int
:: Minute    :== Int
:: SystemMap :== [ExternalSystem]
:: ExternalSystem     = { name2 :: String,
                          url :: String }
:: Request            = INeedADoctor | INeedAPsychologist
:: Appointment        = App Month Day Hour Minute
:: Symptom            = Cough | Fever | Headache | Drowsiness
:: PrescriptionPickup = PrescriptionPickup
:: MedicineRequest    = MedicineRequest
:: Medicine           = MedicineA | MedicineB | MedicineC | MedicinePlacebo

//Step 2: Derive datatypes from step 1
derive class iTask     ExternalSystem, Request, Appointment, Symptom,
                       PrescriptionPickup, MedicineRequest, Medicine
derive bimap (,), Maybe

//Step 3: Inputs, Incoming messages and outputs of atomic activities that are
//        not Void
:: ReceiveDoctorRequestIncdoming   = { message :: Request }
:: SendAppointmentInput           = { systemMap2 :: SystemMap }
:: ReceiveSymptomsIncoming        = { message2 :: [Symptom] }
:: ReceiveSymptomsOutput          = { symptoms :: [Symptom] }
:: SendPrescriptionPickupInput    = { systemMap3 :: SystemMap,
                                      symptoms2 :: [Symptom] }
:: SendPrescriptionPickupOutput   = { medicine :: Medicine, action :: String }
:: ReceiveMedicineRequestIncoming = { medicineRequest :: MedicineRequest }
:: SendMedicineInput              = { systemMap4 :: SystemMap,
                                      medicine2 :: Medicine }
:: DoctorPoolTaskInput            = { systemMap :: SystemMap }

//Step 4: Derive data types from step 3
derive class iTask     ReceiveDoctorRequestIncoming, SendAppointmentInput,
                       ReceiveSymptomsIncoming, ReceiveSymptomsOutput,
                       SendPrescriptionPickupInput, SendPrescriptionPickupOutput,
                       ReceiveMedicineRequestIncoming, SendMedicineInput

//Step 5: Define constant for start event data
startSystemMap :: SystemMap
startSystemMap =  [
                    { name2 = "Patient John Doe",
                      url = "www.example.com/johndoe" },
                     { name2 = "Doctor's Office John Doe instance",
                      url = "www.doctorsoffice.com/message/johndoe" }
                  ]

startAppointment :: Appointment
startAppointment = App 12 23 12 00

//Step 6: Define atomic tasks
ReceiveDoctorRequestTask :: ReceiveDoctorRequestIncoming UserId Void -> Task Void
ReceiveDoctorRequestTask _ _ _ = return Void

SendAppointmentTask :: Void UserId SendAppointmentInput -> Task Void
SendAppointmentTask _ user input =
  SendExternalMessage (hd input.systemMap2) startAppointment >>|
  SendInternalMessage "doctor" startAppointment
```

```
ReceiveSymptomsTask :: ReceiveSymptomsIncoming UserId Void -> Task ReceiveSymptomsOutput
ReceiveSymptomsTask incoming user _ =
  return { ReceiveSymptomsOutput | symptoms = incoming.message2 }

SendPrescriptionPickupTask :: Void UserId SendPrescriptionPickupInput
                                          -> Task SendPrescriptionPickupOutput
SendPrescriptionPickupTask _ user input =
  SendExternalMessage (hd input.systemMap3) (doctor input.symptoms2) >>|
  return { medicine = doctor input.symptoms2, action = "can be picked up" }
where
      doctor :: [Symptom] -> Medicine
      doctor []        = MedicinePlacebo
      doctor [ Cough ] = MedicineA
      doctor [ Fever ] = MedicineB
      doctor _         = MedicineC

ReceiveMedicineRequestTask :: ReceiveMedicineRequestIncoming UserId Void -> Task Void
ReceiveMedicineRequestTask incoming user _ = return Void

SendMedicineTask :: Void UserId SendMedicineInput -> Task Void
SendMedicineTask _ user input =
  SendExternalMessage (hd input.systemMap4) input.medicine2

// Extra debugging functions that intercept messages or simulate reception of
// messages. This is basically the part at which the message handler can be
// plugged in
SendExternalMessage :: ExternalSystem message -> Task Void | iTask message
SendExternalMessage es message =
  "messagehandler" @:("Outgoing message detected",
                     (showMessageAbout es.name2 message)) >>| return Void

SendInternalMessage :: UserId message -> Task Void | iTask message
SendInternalMessage receiver message =
  "messagehandler" @:("Internal message detected",
                     (showMessageAbout receiver message)) >>| return Void

ReceiveExternalMessage :: ExternalSystem -> Task messageType
                                                         | iTask messageType
ReceiveExternalMessage es =
  "messagehandler" @:("Incoming message needed", enterInformation es.name2)

//The actual task to be added to the engine
DoctorPoolTask :: Void UserId DoctorPoolTaskInput -> (Task Void)
DoctorPoolTask _ user input =
  ReceiveExternalMessage (hd input.systemMap) >>=
  \wantToSeeDoctor -> ReceiveDoctorRequestTask wantToSeeDoctor user Void >>|
  SendAppointmentTask Void user { systemMap2=input.systemMap } >>|
  ReceiveExternalMessage (hd input.systemMap) >>=
  \receivedSymptomsMessage ->
    ReceiveSymptomsTask receivedSymptomsMessage user Void >>=
  \receivedSymptoms -> SendPrescriptionPickupTask Void user
    { systemMap3=input.systemMap, symptoms2=receivedSymptoms.symptoms } >>=
  \medicineOutput -> ReceiveExternalMessage (hd input.systemMap) >>=
  \medicineRequestMessage ->
     ReceiveMedicineRequestTask medicineRequestMessage user Void >>|
  SendMedicineTask Void user { systemMap4=input.systemMap,
                             medicine2=medicineOutput.medicine }

Start :: *World -> *World
Start world = startEngine workflowitems world
where
      workflowitems = [ workflow "PoolTask"
                          (DoctorPoolTask Void
                                          "root" { DoctorPoolTaskInput |
                                               systemMap = startSystemMap } )
                      ]
```

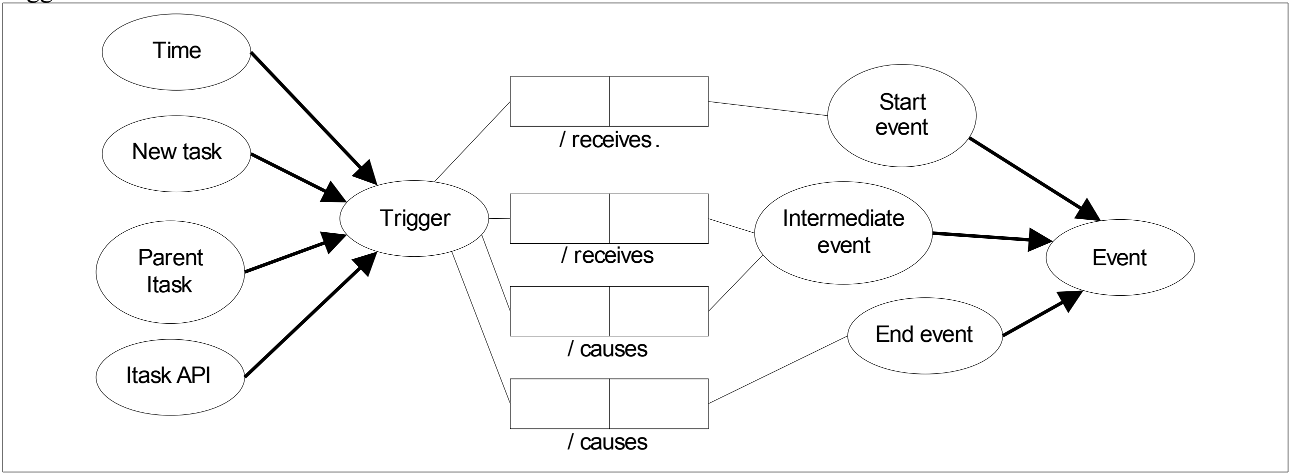
The three extra functions that delegate to the message handler can be explained as follows:

Function	**Description**
SendExternalMessage	Is supposed to send a message to an external system. Could be implemented by calling a web service on the external system as shown with the RPC daemon before.
ReceiveExternalMessage	The current system is waiting for an external message from outside. It could be that the message has already been sent before this function is called and therefore a mailbox solution would be preferable here to store the message.
SendInternalMessage	A message is sent to an actor somewhere inside the Itasks system. The real implementation could use the delegation function of the Itasks system as demonstrated before, but for debugging purposes this message is just printed to the messagehandler user.

All three functions are defined in a polymorphic way, which makes it possible to have them work with any data type the Itasks system can deal with.

6.7 Implementing triggers in Itasks

Recall that the concept of triggers has been introduced as follows in the meta model, which means four different kind of triggers need to be covered:



6.7.1 The time trigger in Itasks

For the time trigger, several solutions among the following ones are possible:

- An RPC call to a web service is made, which returns a value after a certain amount of time has passed or a certain time on the remote system's system clock has been reached (This requires an abstraction from the latency times of the communication, which in most business settings should not be a problem since they are not realtime systems). This solution makes it possible to have different systems synchronize with a central time server.
- The just mentioned option can also be implemented using the time and date functions the Itasks system offers by default. These are explained in Appendix D among the other Itasks constructs.

To implement the second solution above (note that the first one is done in a similar way), the first building brick will be a task with a deadline. This one is based on the atomic task. The result of the task is of form (Maybe a), with 'a' being the normal data type of the task. When the task succeeds in time, the result is of form (Just a) and when the task does not finish in time the result will be (Nothing). This leads to the following code:

```
module deadlinetask

import iTasks, CommonDomain

:: DeadLineTaskInput = { message :: String }
:: DeadLineTaskOutput = { number :: Int }

derive class iTask DeadLineTaskInput, DeadLineTaskOutput, Either
derive bimap (,), Maybe
```

```
DeadLineTask :: UserId DeadLineTaskInput Time → Task (Maybe DeadLineTaskOutput)
DeadLineTask user input timer =
  eitherTask
     (user @:("Perform task DeadLineTask please",
          showStickyMessageAbout "" input ||- (enterInformation "")))
     (waitForTimer timer)
  >>= \result -> parseEitherResult result
where
  parseEitherResult :: (Either DeadLineTaskOutput Void)
                                      -> Task (Maybe DeadLineTaskOutput)
  parseEitherResult (Left value)    = return (Just value)
  parseEitherResult (Right _ )      = return Nothing

Start :: *World -> *World
Start world = startEngine workflowitems world
where
      workflowitems = [ workflow "SpecificAtomicTaskNM"
                          ( DeadLineTask "root"
                                         { message = "Enter a number please" }
                                             { hour=0, min=0, sec=10 } >>=
                              \output -> showStickyMessageAbout "" output )
```

The above code leads to a task in which the user is prompted to enter an integer. When the user manages to do this within ten seconds, the result of the whole task will be (Just a), with 'a' the number the user entered. Otherwise the task will finish after 10 seconds with result Nothing. The above task can be reused in the Foreach setting. Implementing the intermediate event in the Foreach code can be done as follows:

```
CompoundTaskNM :: UserId [CompoundTaskNMInput] Time ->
                                      Task (Maybe [CompoundTaskNMOutput])
CompoundTaskNM user inputList timer =
  eitherTask
      sequence "" (map mapAction inputList)
      (waitForTimer timer)
  >>= \result -> parseEitherResult result
where
    mapAction input = BusinessProcessNM user input
    parseEitherResult :: (Either CompoundTaskNMOutput Void)
                                     -> Task (Maybe CompoundTaskNMOutput)
    parseEitherResult (Left value)    = return (Just value)
    parseEitherResult (Right _ )      = return Nothing
```

The above gives either a full list with the results of the subtasks of the Foreach loop, or returns completely nothing. The semantics of the BPMN with the Foreach can be interpreted that way, but also alternatives are possible which can be generated with Clean software as well:

- When the timer runs out a list containing the iterations that have finished in time is returned: This is achieved by putting a deadline on every single iteration and once the deadline has passed, everything but the last iteration (which is a Nothing value) is returned. The return type of the whole activity then remain Task [CompoundTaskNMOutput], but the length of the output list might be smaller then the input list due to unfinished iterations.
- Every single iteration has a time limit: In this case every iteration is handled by an EitherTask with a time limit and some of the results might be nothing. In this case the return type is Task [Maybe CompoundTaskNMOutput]. In case of example of accessing different suppliers, this option is very realistic, since they might not all reply in time or at all.

The first alternative is solved in the following way, using a recursive definition:

```
CompoundTaskNM :: UserId [CompoundTaskNMInput] Time -> (Task [CompoundTaskNMOutput])
CompoundTaskNM user [] timer   = return [] // no work left
CompoundTaskNM user [input:inputList] timer
| deadLineTaskResult == Nothing = return [] // time is up
| otherwise                     = CompoundTaskNM user inputList timer >>=
                                        \result -> return [deadLineTaskResult:result]
where
      deadLineTaskResult = DeadLineTask user input time
```

While the second one looks like this:

```
CompoundTaskNM :: UserId [CompoundTaskNMInput] Time -> (Task [Maybe
CompoundTaskNMOutput])
CompoundTaskNM user inputList  = sequence "" (map mapAction inputList)
where
	mapAction input = DeadLineTask user input time
```

In this second example one could choose to remove all the Nothing values from the list, but the downside is that a certain result can possible not be matched to the corresponding input anymore, due to the shifting of the indexes of the list. The different approaches taken, could also be combined, leading to output type (Task Maybe [Maybe CompoundTaskNMOutput]).

6.7.2 The new task trigger in Itasks

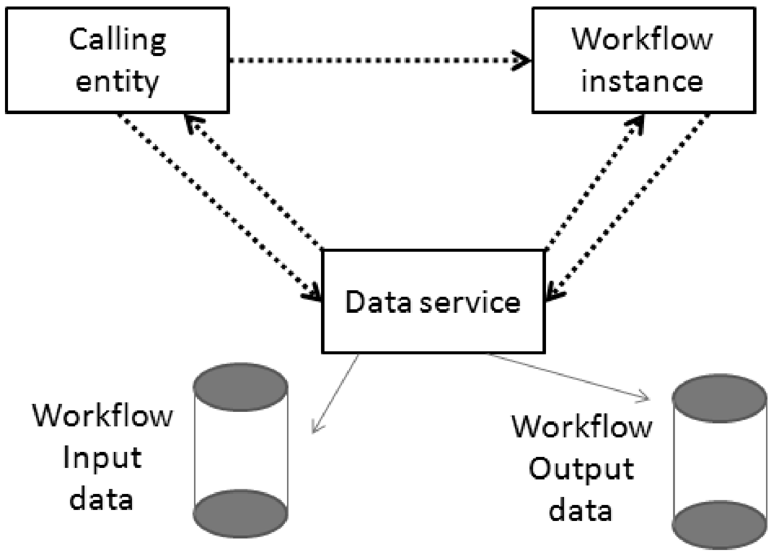
A new task trigger (which is actually triggering a workflow, a decorated task) is always associated with a start event, since this is the point at which a new task starts running. A new task can be started in two different ways in the Itasks environment, namely:

- User click: A user can double click on a process in the Itasks environment on the left hand side, after which a instance of this process is created.
- Handler: The just mentioned user click will actually use this handler. What happens is that whenever a REST call is made (opening a parametrized URL linking to some location on the Itasks system, see Appendix G), a new instance is created. This could happen from any location in the world.

The downside of the Itasks system right now, is that the trigger of new tasks can not be done with parameters and that the type of the workflow (a decorated task) needs to be of (Task Void). This makes it at first sight impossible to have information flows going through them and so far this has been solved in the examples by having fixed parameters and printing information to the screen before the workflow ends. In reality this is not a satisfactory solution and therefore the following algorithm in pseudo code is introduced here:

```
1) The calling entity puts its parameters at some place which is accessible to for workflow
2) The workflow in Itasks is triggered, but is wrapped with an activity in the beginning that picks up the parameters from the just noted shared data resource
3) The workflow is executed
4) The result of the workflow is written to the shared data resource
5) The workflow ends and triggers the calling entity with a Void result
6) The calling entity can pick the result of the workflow at the shared data resource
```

In concrete this leads to the following diagram, which shows what is going on:



The data service could be implemented using a web service and regulates access to the data sources. The service could then be part of the workflow as a fixed parameter in the code.

6.7.3 The parent task trigger in Itasks

When a task is decomposed in smaller tasks, the parent task triggers a subtask when it is included in the code. Implicitly this means that the start event of this smaller task is triggered. When the smaller task is done and returns a value, it causes a trigger for the main task to resume its activities again. In addition to the successful ending of a smaller task, Itasks offers the concept of an exception in a similar way JAVA does. The following functions are relevant for the implementation of exceptions:

```
/**
* Exception combinator.
*
* @param The normal task which will possibly raise an exception of type e
* @param The exception handling task which gets the exception as parameter
* @return The combined task
*/
try  :: !(Task a) !(e -> Task a) -> Task a | iTask a & iTask e

/**
* Exception throwing. This will trough an exception of arbitrary type e which
* has to be caught by a higher level exception handler combinator.
*
* @param The exception value
* @return The combined task
*/
throw :: !e -> Task a | iTask a & TC e
```

Whenever an exception occurs, the second part of try is a function that takes an exception and still returns something of type (Task a), despite that the exception that occurred.

6.7.4 The Itask API trigger

In this sub paragraph the communication of the Itasks system with its environment is discussed. Appendix G shows some constructs that are currently available in the Itasks system, but at the time of writing there is no well defined and documented API available yet. This became clear when pools including messaging functionality between different systems was needed, and the inability to inject a received message into a task was a consequence. This sub paragraph deals with all the triggers and events in BPMN that involve an external system, independent of what the Itasks system supports or not. Since the new task trigger has already been discussed before it is omitted from this sub paragraph. Other triggers that play a role are:

- Triggering an entity in the environment: This can be done using an RPC call or web service. Normally the goal of this trigger is to create a side effect outside the Itasks system. An example is sending an SMS using Voipbuster[R17] to someone or starting some external process which possibly is described in BPMN as well. The result of the call is sent back by the RPC Deamon using an Itask handler.
- Receiving a message from another pool: The previous paragraph showed that whenever pools are used the ability to inject data in a task by entities outside the Itasks system is needed. Currently only the enterInformation statement offers this ability, which is however assumed to only be accessed by human actors.

6.8 A generic algorithm to map the extended BPMN meta model to Itasks

In the previous chapters code has been generated, which supposedly should be generated automatically. In order to make this assumable pseudo code has been given now and then and the code has been annotated with the different steps. Also not always the shortest or most elegant code has been generated, mostly in order to illustrate that when the meta model population is known, the code can be generated without using advanced constructs or calculations.

This paragraph takes all the pseudo code from before and mixes everything together (exploiting the overlap among them) in order to come to a general algorithm in pseudo code to make a transformation between a BPMN meta model and executable code in Clean (using the Itasks library) representing the same semantics. This leads to the following pseudo code:

```
1) Define all the data types that are going to be used
2) Derive gPrint, gVisualize, gParse and gUpdate (class iTask) for all the data types from step 1
3) Define all the inputs and outputs of the activities that are going to be used
4) Derive gPrint, gVisualize, gParse and gUpdate (class iTask) for all the data types from step 3
5) Create fixed input data for activities that are first in a sequence flow
6) Define all the atomic activities and chose one of the four proposed implementations for all of them, simply by cutting and pasting the introduced code
7) Build compound activities and sequences depending on how the BPMN diagram is organized. The input and output of the activities can be determined by looking at the data layer of the BPMN diagram.
8) Decorate the task that is supposed to represent the whole business process to upgrade it to a workflow. Use the input data from step 5 as parameters
9) Create some lines of code to initialize the Itasks engine and add the workflow from step 8 to it
```

The leads to Clean code that can be compiled and ran in order to implement the BPMN system one wishes. Instead of doing so, alternative approaches are considered in the next paragraph, each having their advantages and disadvantages.

6.9 Design decisions taken during the mapping from the extended BPMN meta model to Itasks

In the current chapter, everything has been about generating Clean code that can be run using an algorithm that has the BPMN meta model population as input. This could for example be done in the Clean language itself, but also transformation languages such as Stratego/XT[58][R40], TXL[R41], Tefkat[52][R42] and VIATRA[26][R43] might turn out to be useful for this process. The downside of the transformation approach is that one has to generate the code first and then run it through the compiler before any result can be seen. This makes the approach inflexible for changes. Since this chapter has mostly been meant as a proof of concept, this is not really a problem. But it is still worth to point out the alternatives one has to this approach, which are:

- Feeding the result to an interpreter: Although slower then passing it through a compiler, interpreting code at runtime is possible in a relatively fast way for the Clean language[15]. An interpreter could take the generated code as input and execute right away after generation. This makes it possible to plug in additions or changes to the workflows later on and does not call for two distinct steps (generating and compiling) anymore. Another advantage of this approach is that it does not require much change to the initially taken approach in this chapter. The downsides might be sacrificing performance and having less confidence in the quality of the program (since the initial obligated compile step before hand makes sure many mistakes are eliminated before the program is even started).
- Defining everything at meta level: In this case Clean software is written which takes the meta model population as input, and dynamically generates everything needed as a consequence. For example: There might be a function that takes a list of activities (in String format from the meta model) as input, and returns actual instances of activities. This is possible in the Clean language because it is a higher order language. This approach offers two advantages. First of all everything can happen dynamically which makes the program flexible with respect to changes and additions. Second this approach results in code which describes the semantics of the meta model very carefully, since a direct translation needs to be made from the meta model to something executable. As a disadvantage this approach requires a complete software package in order to test or show something, which makes it hard to do so. Unlike the small code examples that are described before, it is very hard to semantically describe the meta model in an incremental way, because even the smallest examples involve almost all the concepts of the BPMN meta model. The other disadvantage is that type checking and seeing if everything fits well, need to happen at runtime. This requires a lot of extra software that investigates the meta model to do so, while the Clean compiler normally offers this functionality for free.

However, the main idea behind this chapter is to show by providing small software examples that the concepts that have been chosen in the BPMN meta model are valuable and complete. This has been achieved by transforming the population of the meta model in a systematic way to executable Clean code, of which its functioning is testing in the next chapter. The alternative approaches, although sometimes more elegant, would have cost more work and possible pushed the concept of what is really going on to the background.

For messaging between pools it turned out that Itasks did not offer the complete functionality needed. A mailbox metaphor has been introduced to deal with the reception of a message, and made the receiver responsible for storing it. This has been done in order to make the sender able to continue its work after the message has been sent. Instead of this approach, an approach with a parallel waiting task to receive the message could have been chosen as well, which makes it possible to only receive the message when it is really needed by the receiver. This approach has not be chosen, because it conceptually almost the same, but does increase complexity.

6.10 Conclusion

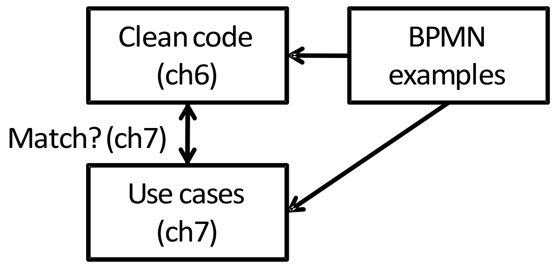
This chapter has taken the extended BPMN meta model (the meta models of the last two chapters combined) as a starting point. When this meta model is populated by providing a BPMN diagram and defining the several extensions (data layers, implementation of atomic activites), it becomes possible to define how to map it to Clean code using the Itasks library. This definition is given in pseudo code in this chapter, after concrete implementation in Clean is given for the different BPMN constructs. Also a generic pseudo algorithm is given at the end, which should make it possible to transform a combination of constructs. The executable code has been compiled and briefly tested, but in order to have confidence in its behaviour, the next chapter will focus on testing the implementations.

7 Validating the mapping to Itasks

The last chapter included the creation of a lot of code. The code that has been created is supposed to represent an implementation of the BPMN diagram that makes sense semantically. In order to check this the following steps will be taken for each piece of software:

– A recap of the BPMN diagram including data flow and the provision of some informal comments about it to illustrate what is going on when needed.
– Creation of a use case that describes the interactions between the system and all actors involved, which is created intuitively by looking at the BPMN diagram in a common sense but structural way.
– Running one or more scenarios of the code, to show what it is really doing.
– Comparing what is happening with the use case(s) and finding out whether there is a match.

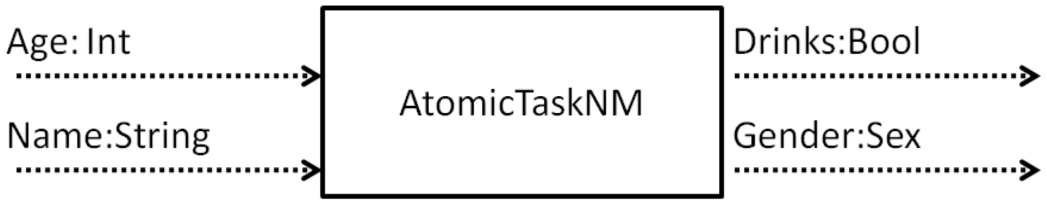
In diagram form this gives the following picture (which is a subset of the initial overview, given at the beginning of this thesis):



In addition to this it is worth to add that the code that has introduced before already passed the compiler checks of Clean, which means basically a lot of technical checks (such a strong typing, syntax and executability) have already been verified.

7.1 Validating the mapping of atomic activities

For atomic activities the following single unit BPMN diagram (including data flow) was introduced before:

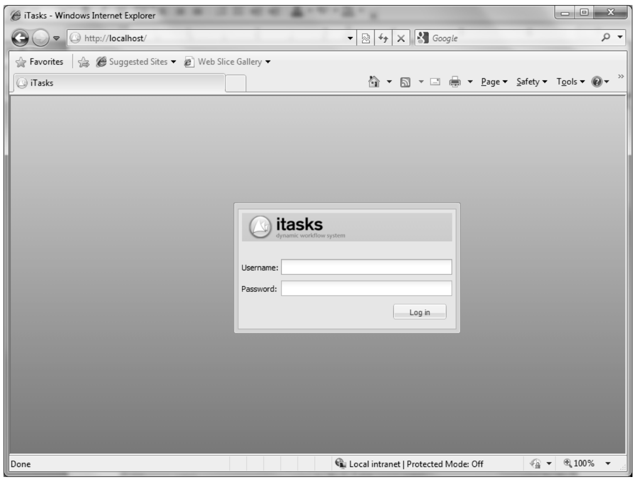


7.1.1 Validating the mapping of user interaction

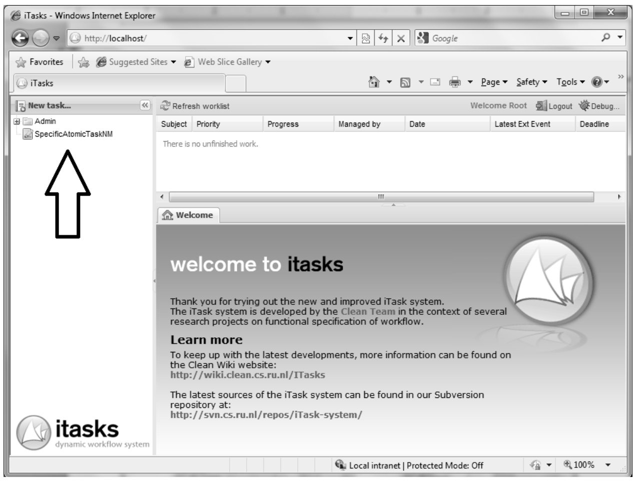
The previous chapter showed an implementation for atomic activities as a 'User interaction'. The use case for this implementation is as follows:

Use case Atomic Activity User interaction	
Name	Atomic Activity User interaction
Summary	The Atomic Activity transforms data of one kind to data of another kind. Using the User interaction means that a user will be taking care of this transformation.
Actors	Root
Basic course of events	1) System provides input data of the atomic activity to the Root 2) Root provides output data to system
Exceptions	2a) Root provides data to system that does not meet the output data criteria
Alternative paths	N/A

When the code is executed and http://localhost/ is executed from the browser, the result should be something that matches the above use case in order to validate the right semantics. The following four step scenario occurs (of which the first two steps deal with initializing the activity):


Username:
Password:

1: Login screen of Itasks. The credentials of Root are entered here.


There is no unfinished work.

welcome to itasks

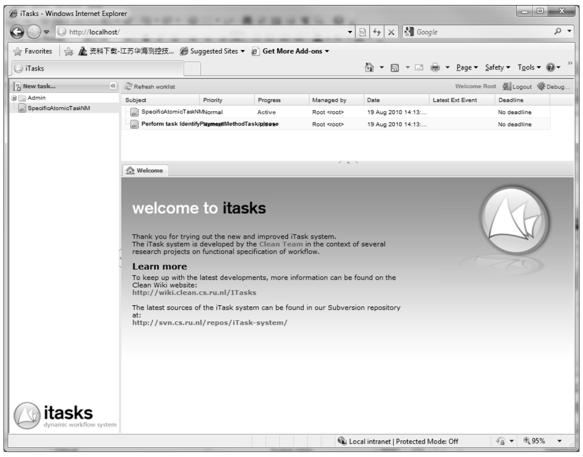
Thank you for trying out the new and improved iTask system.
The iTask system is developed by the Clean Team in the context of several research projects on functional specification of workflow.

Learn more

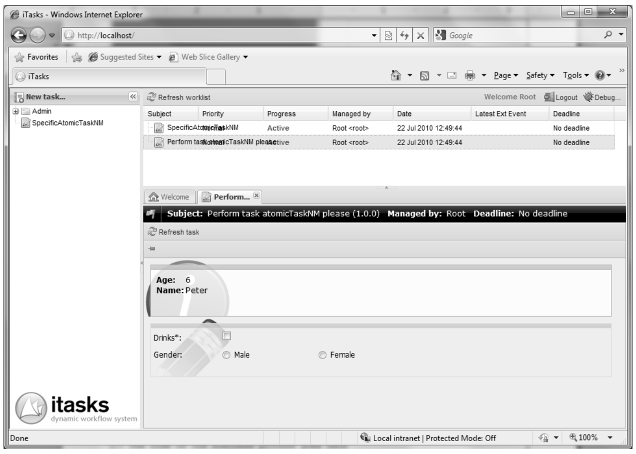
To keep up with the latest developments, more information can be found on the Clean Wiki website:
http://wiki.clean.cs.ru.nl/ITasks

The latest sources of the iTask system can be found in our Subversion repository at:
http://svn.cs.ru.nl/repos/iTask-system/

2: The Itasks system's personal workspace shows up. Double clicking on the SpecificAtomicTaskNM is done to create a new instance


Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline
SpecificAtomicTaskNM	Normal	Active	Root <root>	19 Aug 2010 14:13:...		No deadline
Perform task IdentifyPaymentMethodTask please			Root <root>	19 Aug 2010 14:13:...		No deadline

3: The instance appears in the to do list.


Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline
SpecificAtomicTaskNM		Active	Root <root>	22 Jul 2010 12:49:44		No deadline
Perform task atomicTaskNM please		Active	Root <root>	22 Jul 2010 12:49:44		No deadline

Subject: Perform task atomicTaskNM please (1.0.0) Managed by: Root Deadline: No deadline

Age: 6
Name: Peter

Drinks*:
Gender: Male Female

4: The instance is opened, showing the activity's input and requested output fields that need to be filled in. After doing this, okay can be pressed and the activity is done.

In step 4 the input gates are shown, namely age (valued 5) and name (valued "Peter"). The user can enter the output gates drinks and gender and what can be seen is that the interface for entering them already pretty much shows the type. In other words: depending on the data type the Itasks system is expecting, a way of entering (a check box for Booleans, a text field for Strings, a selection box for multivalued data types such as Gender) is automatically enforced. Another aspect that is worth mentioning is that changes are always stored directly. This means that if for example the male gender is selected (as in the above picture), and the system happens shut down and is restarted again, the selection of that value is not lost.

The reason the value is not lost is caused by everything that is stored by the Itasks system. From the perspective of this thesis this storage at instance level could be interesting as a potential addition to the meta model. The meta model so far has not taken instances of business processes into account but only deals with the generic level in which work is done. See Van der Aalst's paper[3] for more about this. This meta model of instances is purposely left outside the scope this thesis, since the Itasks system offers a gentle way to deal with it.

In the above picture the work list shows 2 entries, while only task instance has been generated. This is because of the @: construct for delegating that has been used. One item encompasses the task as just mentioned, while the other item shows the progress of the task and whom it has been delegated to. Normally these two tasks are in the work list two different people (which is illustrated in the pools and lanes paragraph), but over here they are the same person. In fact, in this example the delegate does not really offer extra value, but is already added in anticipation to the more advanced cases that are about to come.

With respect to the use case one can state that BCOE step 1 and 2 are rightfully implemented. The exception path is not even possible in the implementation, because the UI as just shown does not even allow for the user to enter something that is not output data. In the above scenario a log in as Root user took place, which means that the interaction happened with the right actor.

7.1.2 Validating the mapping of an abstraction

For the abstraction, also known as decomposed business process implementation, no concrete example has been given. Since one does not know how the composition looks like from the black box point of view. This causes it to be impossible to show through interactions. The next paragraph shows how sequential business processes are made and the paragraph after that will deal with compound ones. This knowledge can be used to wrap those activities in something to create an new atomic activity. Since the steps of sequential and compound activities will be validated, the decomposed business process can be considered to be defined in a valid way as well as a consequence.

7.1.3 Validating the mapping of the other atomic activities

This subparagraph deals with the other options to implement an atomic activity, namely 'Calculation', 'External interaction' and 'Data access'. These three atomic activities deal with an automated interaction and therefore transform input to output without human interference (unless the implementation of the web service does so at a remote system). The use case for all of them is as follows:

Use case Atomic Activity Calculation/External interaction/Data access	
Name	Atomic Activity Calculation/External interaction/Data access
Summary	The Atomic Activity transforms data of one kind to data of another kind. When Calculation/Internal interaction/Data access is used, it means the transaction is delegated to a different automated entity, named External system.
Actors	External system
Basic course of events	1) System provides input data of the atomic activity to the External system 2) External system provides output data to system
Exceptions	1a) System tries to provide input data of the atomic activity to the External system, but can not reach it 2a) Go to step 1 / End use case unsuccessfully 2b) External system does not reply for some reason 3b) Go to step 1 / End use case unsuccessfully
Alternative paths	N/A

Step 2a and 3b show two alternatives which are both considered to be valid. Namely retrying the action by redoing the whole use case (since the remote system or the connection to the system might temporarily be malfunctioning) or accepting that the external system is not helpful and aborting the use case. When a number of retries is defined, an approach combining both approaches is chosen.

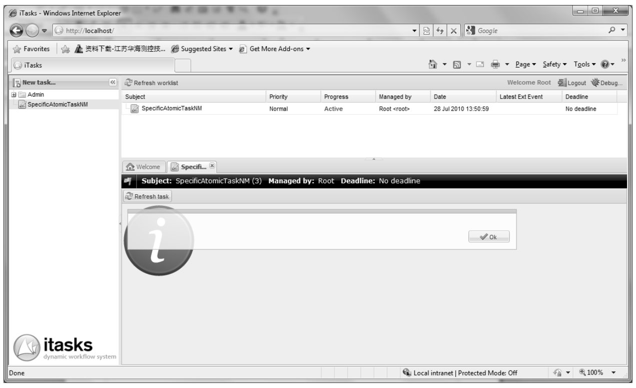
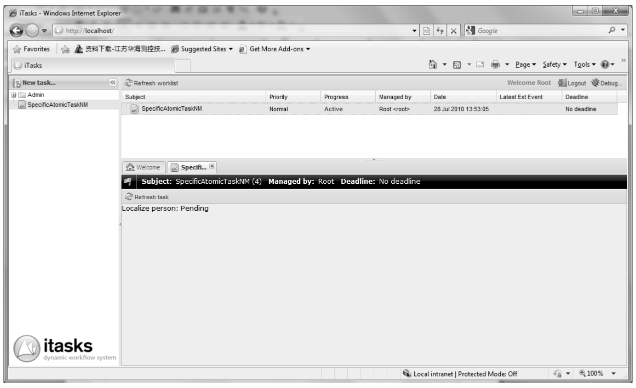
First the 'Calculation' case. For this system the exception 1a does not apply, since the external system is embedded in the Clean code for the Itasks system. This means that it is always reachable. When the sample code of the previous chapter is run with input value { maxPrime = 500}, the result becomes as expected (for those who can enumerate all prime numbers up to 500):

```
(AtomicTaskNMOutput
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,
113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,211,223,227,229,233,
239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,331,337,347,349,353,359,367,
373,379,383,389,397,401,409,419,421,431,433,439,443,449,457,461,463,467,479,487,491,499]
)
```

However exception 2b is still possible. Imagine that the mathematical function is a calculation that takes an infinite amount of time. In that case the whole process will stall, which can be seen as an abort (use case step 3b).

Another case that is worth paying attention to, is a calculation that causes an error (for example a stack or heap overflow when one tries to generate the list of all prime numbers). In this case the Itasks system will unfortunately crash completely. If one wants to avoid this from happening, some error handling as introduced in the previous chapter with try, catch and throw might offer a solution.

Second the 'External interaction' approach. The exception paths need to be considered here, since the internet connection might go down or the server hosting the service might stop working. For exception path 1a (not being able to reach the server), two scenarios are applicable which are as follows:

	
The server is not reachable (which is tested by messing up the URL). The Google Latitude task from last chapter returns an empty String.	The daemon is down, which causes the task to not be picked up and handled. The result is that the task does not finish and remains in a pending state.

SpecificAtomicTaskNM Normal Active Root <root> 28 Jul 2010 13:50:59 No deadline
Subject: SpecificAtomicTaskNM (3) Managed by: Root Deadline: No deadline

SpecificAtomicTaskNM Normal Active Root <root> 28 Jul 2010 13:53:05 No deadline
Subject: SpecificAtomicTaskNM (4) Managed by: Root Deadline: No deadline
Localize person: Pending

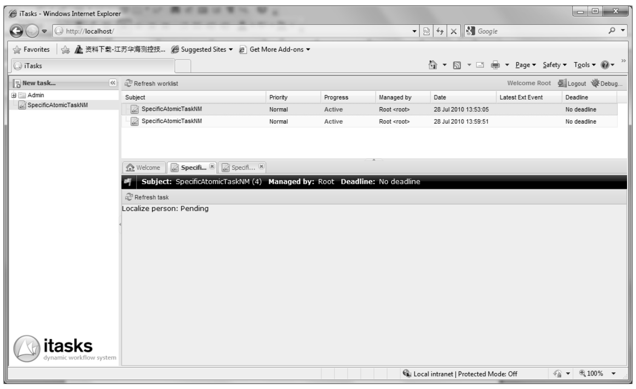
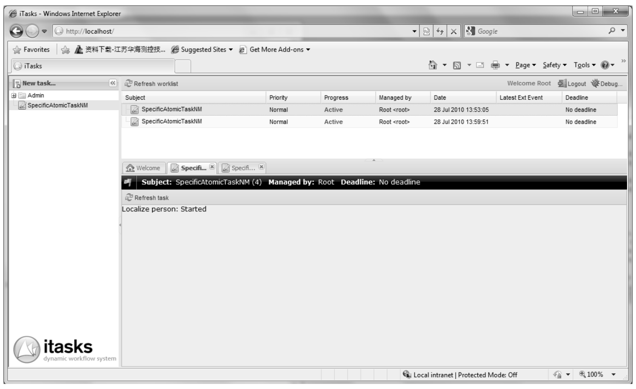
For the left scenario this means that some extra error handling is needed, which either causes the task to return a result that indicates failure, blocks the tasks or just recursively retries the task (possibly for a limited number of times). This can be done by checking the output and depending on it, taking a decision what to do. Another option would be to let the daemon never return something, which causes a scenario similar to the one on the right. The debugging output of the daemon is as follows (which basically means that Apache, a library the daemon uses, cannot open the URL):

```
2010-07-28 14:03:43,086 (WebServiceWorker.java:96) [ERROR]
rpcd.remoteserviceworker.webserviceworker - Exception occurred while executing HTTP
request in WebServiceWorker for task 4.0.0.0
java.net.UnknownHostException: www.thisisafakeserverthatdoesnotexist.com
        at java.net.PlainSocketImpl.connect(Unknown Source)
        at java.net.SocksSocketImpl.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at org.apache.http.conn.scheme.PlainSocketFactory.
                                     connectSocket(PlainSocketFactory.java:123)
```

In the right scenario the task blocks, which matches ending the use case unsuccessfully. At a higher level in the implementation, one needs to be aware of unsuccessful termination and deal with it (for example by defining a time out). When this is not done, one might end up with a lot of instances in the Itasks system which will never end.

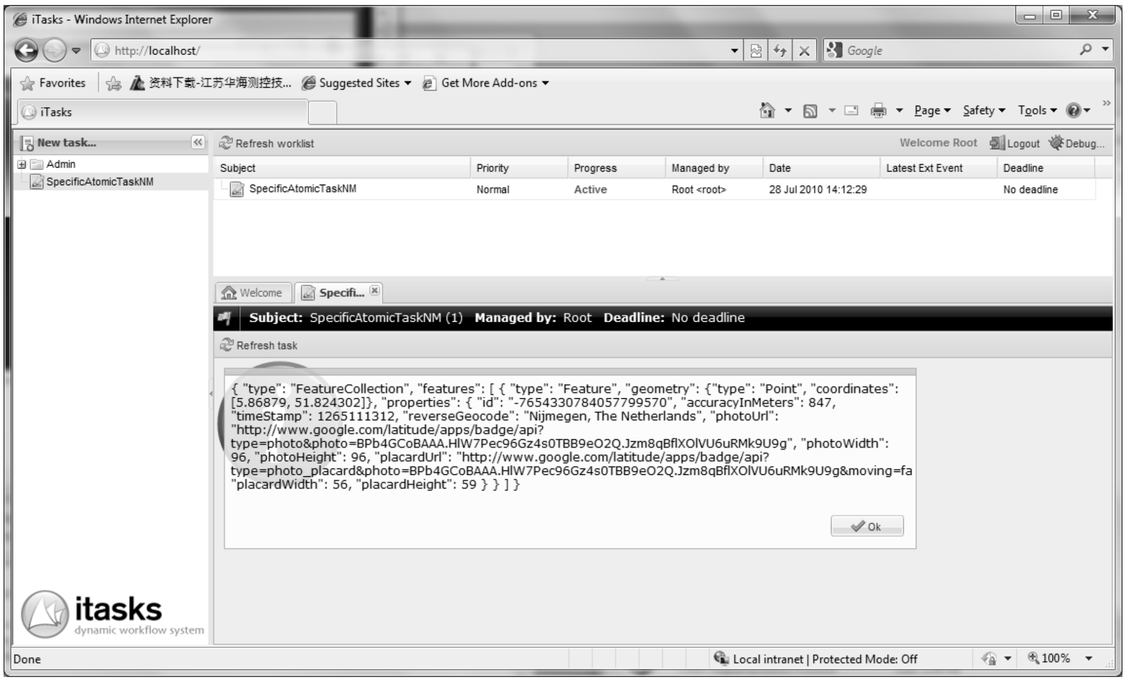
Exception 2b is comparable to the right scenario as shown above, since pending is the initial state of a web service task. Since step 3b of the use case is similar to 2a, and the behavior that the implementation shows as well, step 3b can be considered covered.

The final scenario, for which the code has been provided in the last chapter, and which turns out to be successful looks as follows:

	
1: Task is initiated and is in the pending state.	2. The task goes to the started state when the daemon picks it up.

SpecificAtomicTaskNM Normal Active Root <root> 28 Jul 2010 13:53:05 No deadline
SpecificAtomicTaskNM Normal Active Root <root> 28 Jul 2010 13:59:51 No deadline
Subject: SpecificAtomicTaskNM (4) Managed by: Root Deadline: No deadline
Localize person: Pending

Localize person: Started



Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline
SpecificAtomicTaskNM	Normal	Active	Root <root>	28 Jul 2010 14:12:29		No deadline

Subject: SpecificAtomicTaskNM (1) **Managed by:** Root **Deadline:** No deadline

```
{ "type": "FeatureCollection", "features": [ { "type": "Feature", "geometry": {"type": "Point", "coordinates":
[5.86879, 51.824302]}, "properties": { "id": "-7654330784057799570", "accuracyInMeters": 847,
"timeStamp": 1265111312, "reverseGeocode": "Nijmegen, The Netherlands", "photoUrl":
"http://www.google.com/latitude/apps/badge/api?
type=photo&photo=BPb4GCoBAAA.HlW7Pec96Gz4s0TBB9eO2Q.Jzm8qBflXOlVU6uRMk9U9g", "photoWidth":
96, "photoHeight": 96, "placardUrl": "http://www.google.com/latitude/apps/badge/api?
type=photo_placard&photo=BPb4GCoBAAA.HlW7Pec96Gz4s0TBB9eO2Q.Jzm8qBflXOlVU6uRMk9U9g&moving=fa
"placardWidth": 56, "placardHeight": 59 } } ] }
```

3: Eventually, the expected result is returned as given in the previous chapter.

The above scenario steps can be linked to the use case in the following way:

Use case	1		2				
Scenario	1	2	3				

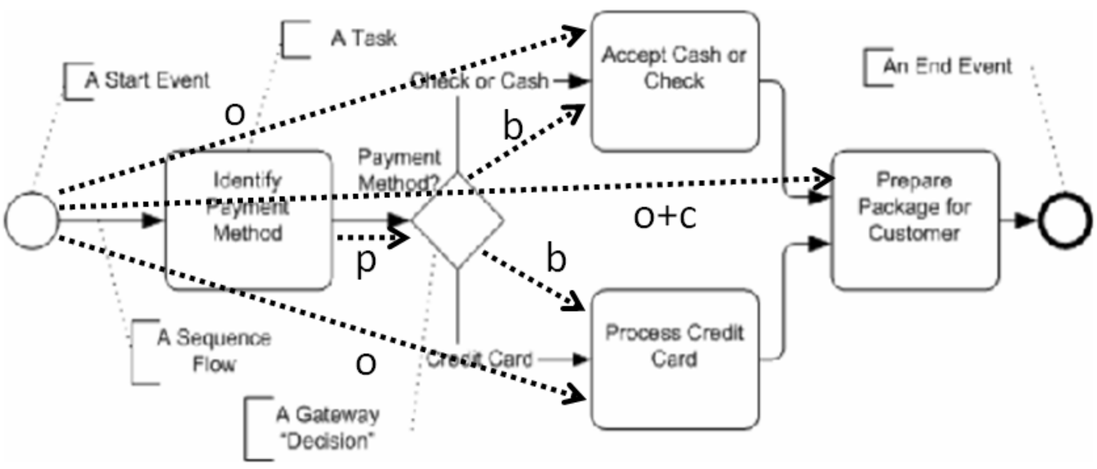
This means that all the use case steps have been covered.

Finally the 'Data access' approach. The functionality is similar to the two previous implementations and the regular case is therefore not considered. When data is read using the 'readDB' command, the command 'mkDBid' always need to be performed first in order to obtain a source to read from. Even when no database exists, mkDBid while create an empty one ad hoc. This causes the data source to always be reachable and also always to return a result (especially when one defines a default value to return in case the input key can not be found).

The only exception case one can imagine, since I/O is the case here (the database is stored on the hard drive in a text file in the previously given example), is that the hard drive has crashed. Since the assumption is that the database storage area is part of the Itask system, this option is not considered. The reason why it is not considered is that one otherwise can make no assumptions about the functioning of the Itasks system at all (eg computer might crash, processor might have a failure in design). Also a slow hard drive is not a problem, since any hard drive is assumed to be faster responding then a human in the 'User interaction', which works fine.

7.2 *Validating the mapping of the sequential and data flow*

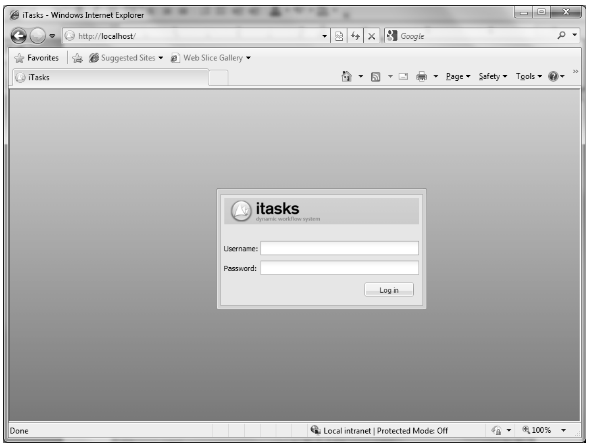
The example code from the previous chapter, based on the BPMN diagram requires three scenarios to be ran in order to test all the traces, namely the Credit Card, Cash and Check case. In addition to this two exceptions that play a role will be mentioned later. Recap that the BPMN diagram including data flow looks as follows:



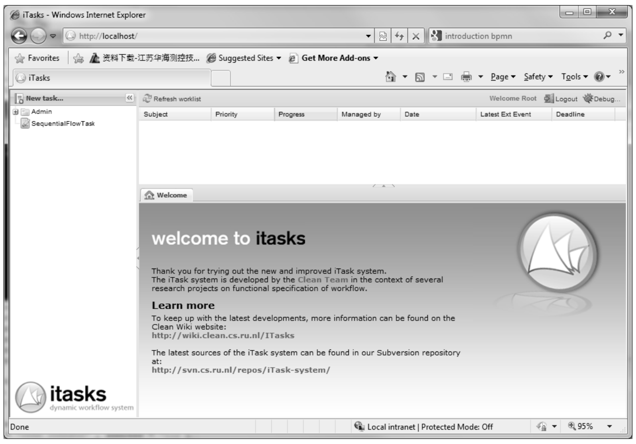
The use case for the sequential flow, which follows from the BPMN diagram, looks as follows:

Sequential activity with gateway	
Name	Sequential activity with gateway
Summary	The sequential activity makes it possible to combine different activities and have them performed after each other in time. The gateway makes it possible to have more then one path, dynamically depending on the contents of the data flow
Actors	Root
Basic course of events	0) The activity is triggered 1) System asks Root which payment method shall be used 2) Root provides the payment method to the system 3) System provides order data to Root 4) Root confirms that a cash or check payment has been made to the system 5) System indicates that a package has to be shipped to a customer with a certain order to Root 6) Root confirms the shipment of the order to the system
Exceptions	4a) Root does not confirm the payment to the system 6a) Root does not confirm the shipment of the order to the system
Alternative paths	4b) Root confirms that a credit card payment has been made to the system 5b) System indicates that a package has to be shipped to a customer with a certain order to Root 6b) Root confirms the shipment of the order to the system

The different scenarios that occur after executing the code are as follows:



1: Log in screen of the Itasks system



welcome to itasks

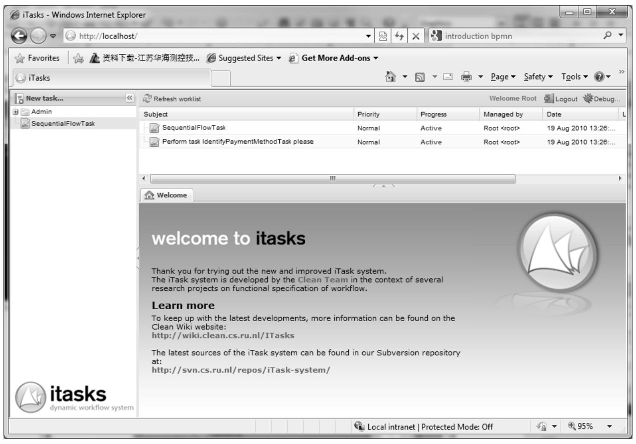
Thank you for trying out the new and improved iTask system. The iTask system is developed by the Clean Team in the context of several research projects on functional specification of workflow.

Learn more

To keep up with the latest developments, more information can be found on the Clean Wiki website: http://wiki.clean.cs.ru.nl/ITasks

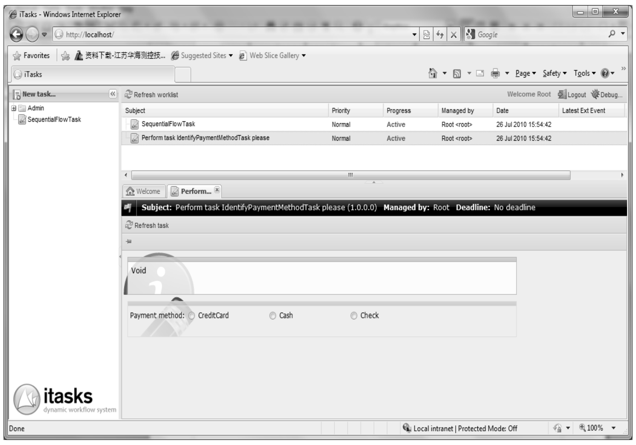
The latest sources of the iTask system can be found in our Subversion repository at: http://svn.cs.ru.nl/repos/iTask-system/

2: Double clicking on the specific task creates a new instance



Subject	Priority	Progress	Managed by	Date
SequentialFlowTask	Normal	Active	Root <root>	19 Aug 2010 13:26:...
Perform task IdentifyPaymentMethodTask please	Normal	Active	Root <root>	19 Aug 2010 13:26:...

3: Instance is shown in the to do list



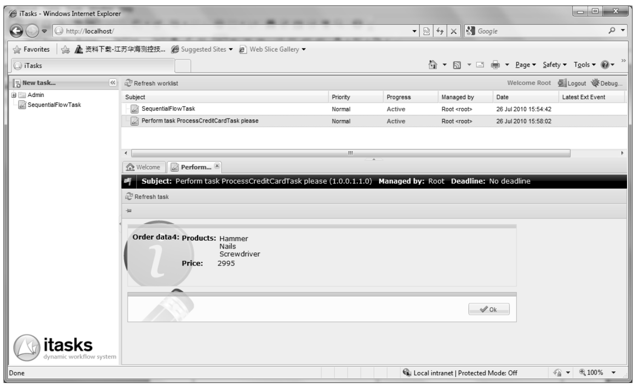
Subject	Priority	Progress	Managed by	Date
SequentialFlowTask	Normal	Active	Root <root>	26 Jul 2010 15:54:42
Perform task IdentifyPaymentMethodTask please	Normal	Active	Root <root>	26 Jul 2010 15:54:42

Subject: Perform task IdentifyPaymentMethodTask please (1.0.0.0) Managed by: Root Deadline: No deadline

Void

Payment method: CreditCard Cash Check

4: Step one of the task is shown, in which a payment method is requested. Depending on the option chosen the following step occurs:
- Credit card (go to step 4a)
- Cash (go to step 4b)
- Check (go to step 4b)



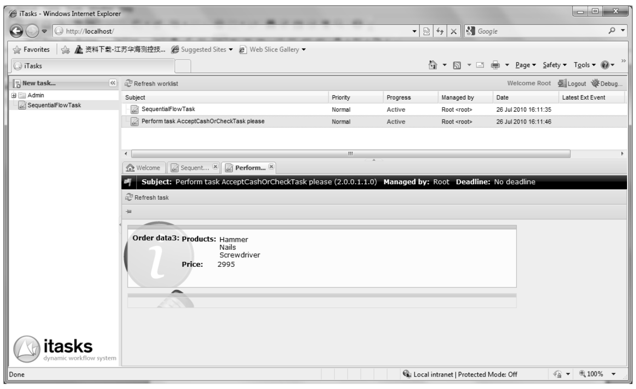
Subject	Priority	Progress	Managed by	Date
SequentialFlowTask	Normal	Active	Root <root>	26 Jul 2010 15:54:42
Perform task ProcessCreditCardTask please	Normal	Active	Root <root>	26 Jul 2010 15:58:02

Subject: Perform task ProcessCreditCardTask please (1.0.0.1.1.0) Managed by: Root Deadline: No deadline

Order data4: Products: Hammer Nails Screwdriver Price: 2995

Ok

5a: A new task ProcessCreditCardTask appears, which shows the order data and asks okay for confirmation. Okay is pressed (go to step 6)

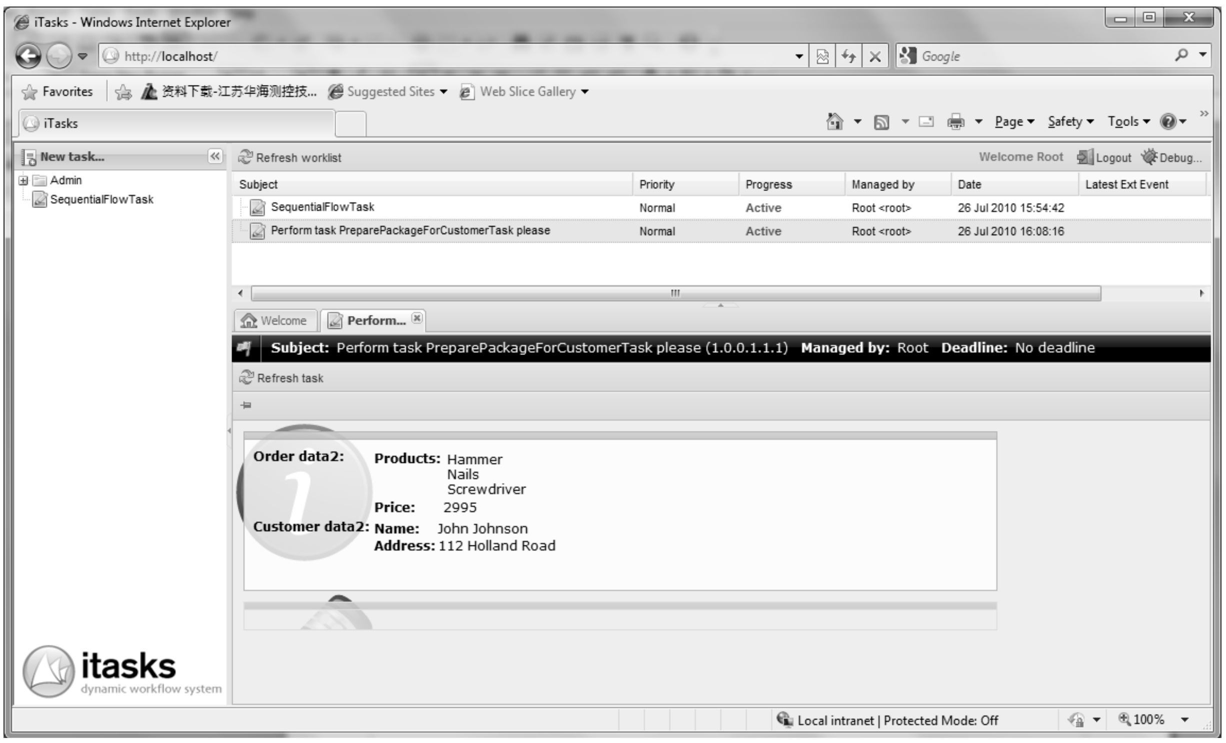


Subject	Priority	Progress	Managed by	Date
SequentialFlowTask	Normal	Active	Root <root>	26 Jul 2010 16:11:35
Perform task AcceptCashOrCheckTask please	Normal	Active	Root <root>	26 Jul 2010 16:11:46

Subject: Perform task AcceptCashOrCheckTask please (2.0.0.1.1.0) Managed by: Root Deadline: No deadline

Order data3: Products: Hammer Nails Screwdriver Price: 2995

5b: A new task AcceptCashOrCheckTask appears, which shows the order data and asks okay for confirmation. Okay is pressed (go to step 6)



Subject	Priority	Progress	Managed by	Date	Latest Ext Event
SequentialFlowTask	Normal	Active	Root <root>	26 Jul 2010 15:54:42	
Perform task PreparePackageForCustomerTask please	Normal	Active	Root <root>	26 Jul 2010 16:08:16	

Subject: Perform task PreparePackageForCustomerTask please (1.0.0.1.1.1) **Managed by:** Root **Deadline:** No deadline

Order data2: **Products:** Hammer
Nails
Screwdriver
Price: 2995
Customer data2: **Name:** John Johnson
Address: 112 Holland Road

6: A new task PreparePackageForCustomerTask appears, which shows the order data, the customer name and asks okay for confirmation. After pressing okay the task is done (which represents that the order has been packed).

What is next is to link these scenarios to the steps of the use case and see if they match. The numbers used in the scenarios and use case are used for identification. This gives the following three traces (of which the check and cash one are similar):

Check:

Use case	0	1	2	3	4	5	6
Scenario	2	4	4	5b	5b	6	6

Cash:

Use case	0	1	2	3	4	5	6
Scenario	2	4	4	5b	5b	6	6

Credit card:

Use case	0	1	2	3	4b	5b	6b
Scenario	2	4	4	5a	5a	6	6

This leaves the two exception cases, which are listed as 4a and 6a in the use case. When a customer does not pay or when an order has not been shipped or never will, the steps that follow in the scenario will not be executed since okay will not be pressed. This is exactly what the use case demands, since it specifies that nothing should happen after step 4a and 6a anymore. If one wants the instances of these cases to disappear after some time, a timeout procedure is an option.

7.3 Validating the mapping of compound activities

In the previous chapter of the implementation, two examples have been shown for compound activities, namely the Euro to RMB converter and the quoting of suppliers example. Both dealt with a Foreach loop.

7.3.1 The currency converter example

For the former example (which has been modified slightly to provide user output) the following use case applies:

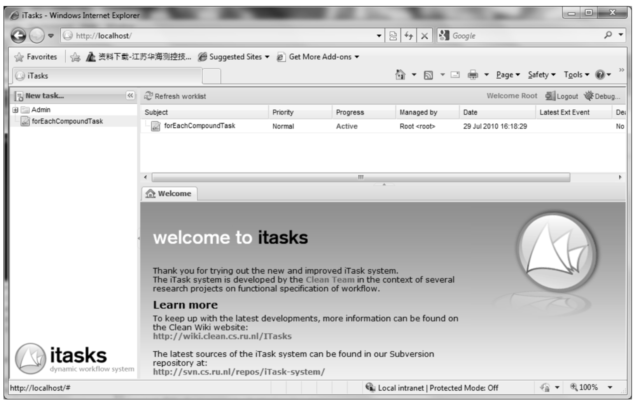
Foreach activity to convert from Euro to RMB	
Name	Foreach activity to convert from Euro to RMB
Summary	The foreach activity performs a mathematical function on a list of elements, after which the results are stored in one task, containing the list of results
Actors	Root, Subsystem
Basic course of events	0) The activity is triggered 1) If there are no amounts to be processed anymore, go to step 5 2) Root provides an amount in Euros to the subsystem 3) The subsystem returns an amount in RMB to the system 4) Go to step 1 5) System provides a list of amounts in RMB to the user
Exceptions	
Alternative paths	

As one can see, another actor named subsystem has been added. The subsystem is responsible for one iteration of the Foreach loop. In order to check whether this works right, a black box approach is not sufficient anymore. By adding some extra debugging code, (which intercepts interactions between the system and subsystem) it becomes possible to look into the system. This is done in the following fashion by modifying the BusinessProcessNM:

```
BusinessProcessNM :: UserId CompoundTaskNMInput -> (Task CompoundTaskNMOutput)
BusinessProcessNM _ input =
  showMessageAbout "Input to subsystem" input
  >>| return { number2 = input.number * 9 }
  >>= \output -> showMessageAbout "Output from subsystem" output
  >>| return output
```

One showMessageAbout intercepts the input to the subsystem and shows it to the Root user, while the other does the same for the output message.

The scenario that then occurs is as follows:



Subject	Priority	Progress	Managed by	Date	Latest Ext Event	De
forEachCompoundTask	Normal	Active	Root <root>	29 Jul 2010 16:18:29		No

welcome to itasks

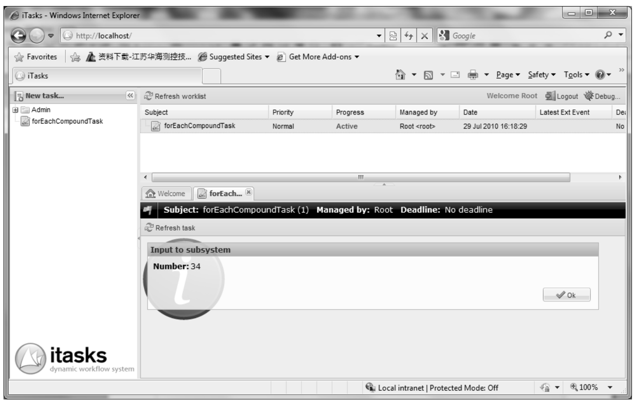
Thank you for trying out the new and improved iTask system.
The iTask system is developed by the Clean Team in the context of several research projects on functional specification of workflow.

Learn more

To keep up with the latest developments, more information can be found on the Clean Wiki website:
http://wiki.clean.cs.ru.nl/ITasks

The latest sources of the iTask system can be found in our Subversion repository at:
http://svn.cs.ru.nl/repos/iTask-system/

1: The task is initiated by double clicking

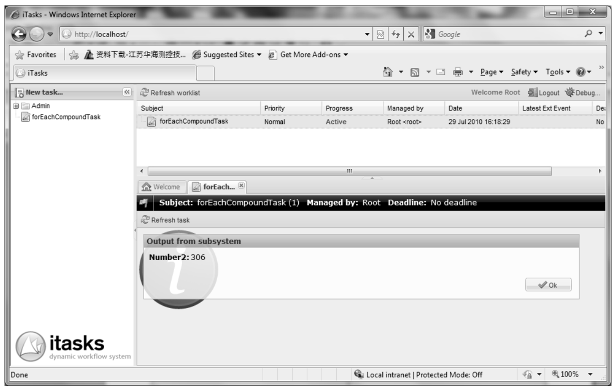


Subject: forEachCompoundTask (1) Managed by: Root Deadline: No deadline

Input to subsystem

Number: 34

2: A message box with title 'input to subsystem' and content { Number = 34 } is shown

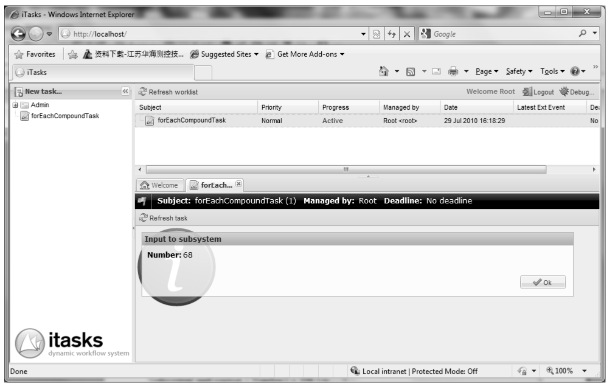


Subject: forEachCompoundTask (1) Managed by: Root Deadline: No deadline

Output from subsystem

Number2: 306

3: A message box with title 'output from subsystem' and content { Number2 = 306 } is shown

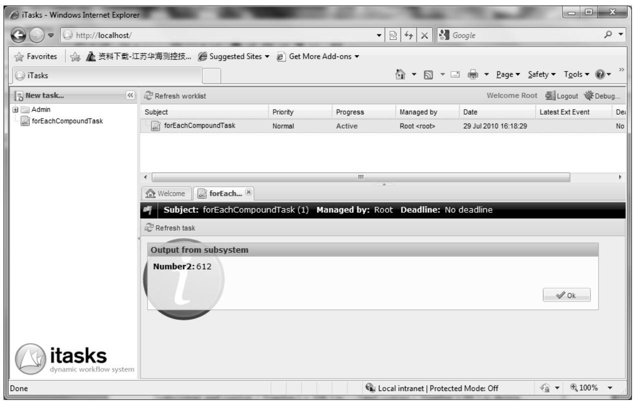


Subject: forEachCompoundTask (1) Managed by: Root Deadline: No deadline

Input to subsystem

Number: 68

4: A message box with title 'input to subsystem' and content { Number = 68 } is shown



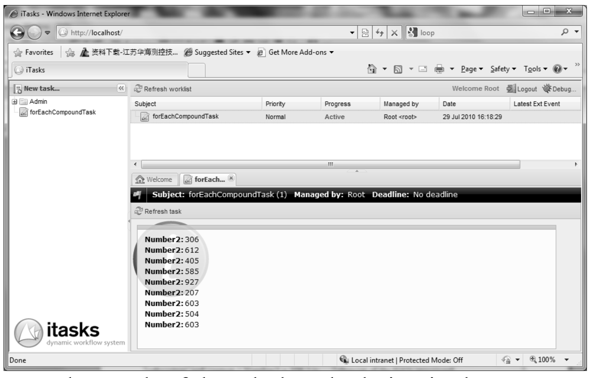
Subject: forEachCompoundTask (1) Managed by: Root Deadline: No deadline

Output from subsystem

Number2: 612

5: A message box with title 'output from subsystem' and content { Number2 = 306 } is shown

6: Step 4 and step 5 are repeated until every element of the list is processed



Subject: forEachCompoundTask (1) Managed by: Root Deadline: No deadline

Number2: 306
Number2: 612
Number2: 405
Number2: 585
Number2: 927
Number2: 207
Number2: 603
Number2: 504
Number2: 603

7: The result of the whole calculation is shown.

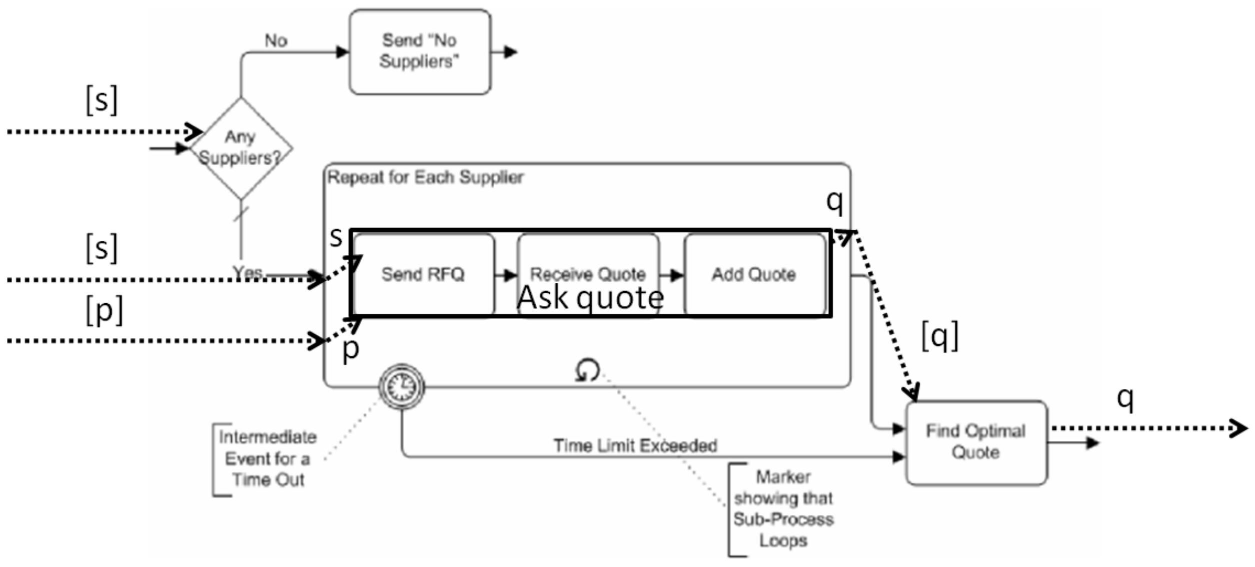
Matching the above scenario with the use case gives:

Use case	0	1	2	3	4	5	
Scenario	1	6	2, 4	3, 5	jump	7	

Once again all the use case steps are covered. The step 4 shows jump in the scenario field, since it is only a goto statement and no work needs to be performed in the scenario.

7.3.2 The supplier quoting example

Next the Foreach example for the supplier quoting. Recall that the BPMN diagram including data flow looks as follows:



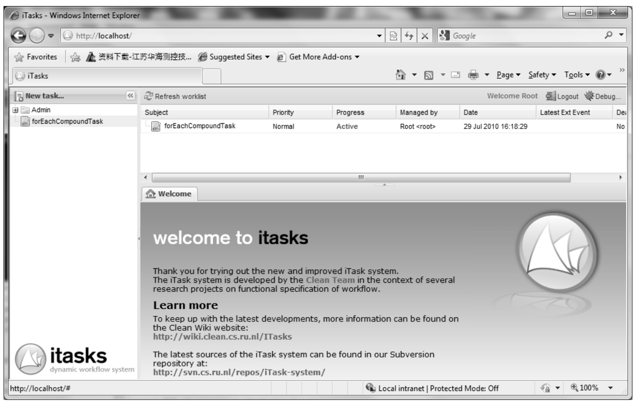
This can be expressed in the following use case (with the timer interruption omitted):

Foreach activity to ask different suppliers for a quote for a certain product	
Name	Foreach activity to ask different suppliers for a quote for a certain product
Summary	The Foreach activity contacts different suppliers and asks them for a quote for a certain product. After this process the quote with the lowest price is selected.
Actors	Root, Subsystem
Basic course of events	0) The activity is triggered 1) If there are no suppliers to be processed anymore, go to step 5 2) Root provides an supplier and product to the subsystem 3) The subsystem returns a quote to the system 4) Go to step 1 5) System returns the most optimal quote to the root user
Exceptions	
Alternative paths	1a) There are no suppliers to be processed at all, go to step 2a 2a) System returns a nothing value to the root user

Once again, in comparison to the code as provided before, the subsystem's interface to the system is intercepted to be able to detect step 2 and 3 of the use case. This leads to the following code:

```
BusinessProcessNM :: UserId CompoundTaskNMInput -> (Task CompoundTaskNMOutput)
BusinessProcessNM _ input =
  showMessageAbout "Input to subsystem" input >>|
  return (receivedQuote input)
  >>= \output -> showMessageAbout "Output from subsystem" output
  >>| return output
```

The scenario in which there are suppliers is as follows:



Subject	Priority	Progress	Managed by	Date	Latest Ext Event
forEachCompoundTask	Normal	Active	Root <root>	29 Jul 2010 16:18:29	

welcome to itasks

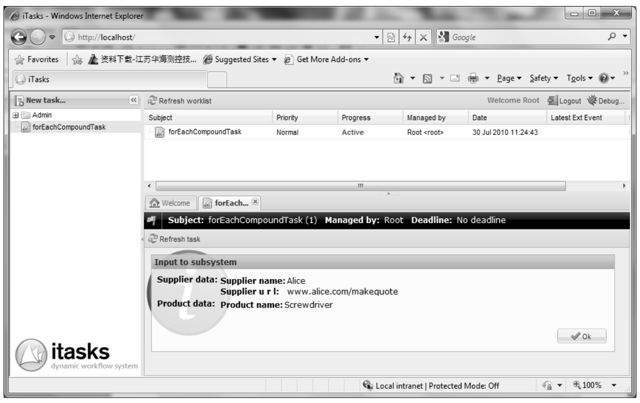
Thank you for trying out the new and improved iTask system.
The iTask system is developed by the Clean Team in the context of several research projects on functional specification of workflow.

Learn more

To keep up with the latest developments, more information can be found on the Clean Wiki website:
http://wiki.clean.cs.ru.nl/ITasks

The latest sources of the iTask system can be found in our Subversion repository at:
http://svn.cs.ru.nl/repos/iTask-system/

1: The task is initiated by double clicking.



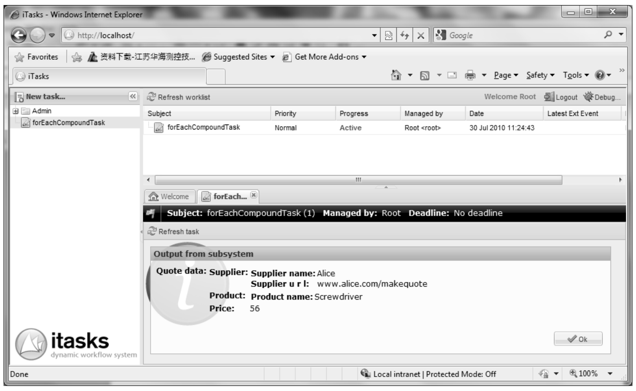
Subject	Priority	Progress	Managed by	Date	Latest Ext Event
forEachCompoundTask	Normal	Active	Root <root>	30 Jul 2010 11:24:43	

Subject: forEachCompoundTask (1) Managed by: Root Deadline: No deadline

Input to subsystem

Supplier data: Supplier name: Alice
Supplier u r l: www.alice.com/makequote
Product data: Product name: Screwdriver

2: A message box with title 'input to subsystem' and content { SupplierData = { Supplier name = "Alice", Supplier URL = "www.alice.com/makequote"}, ProductData = { "ProductName = "Screwdriver" }} is shown



Subject	Priority	Progress	Managed by	Date	Latest Ext Event
forEachCompoundTask	Normal	Active	Root <root>	30 Jul 2010 11:24:43	

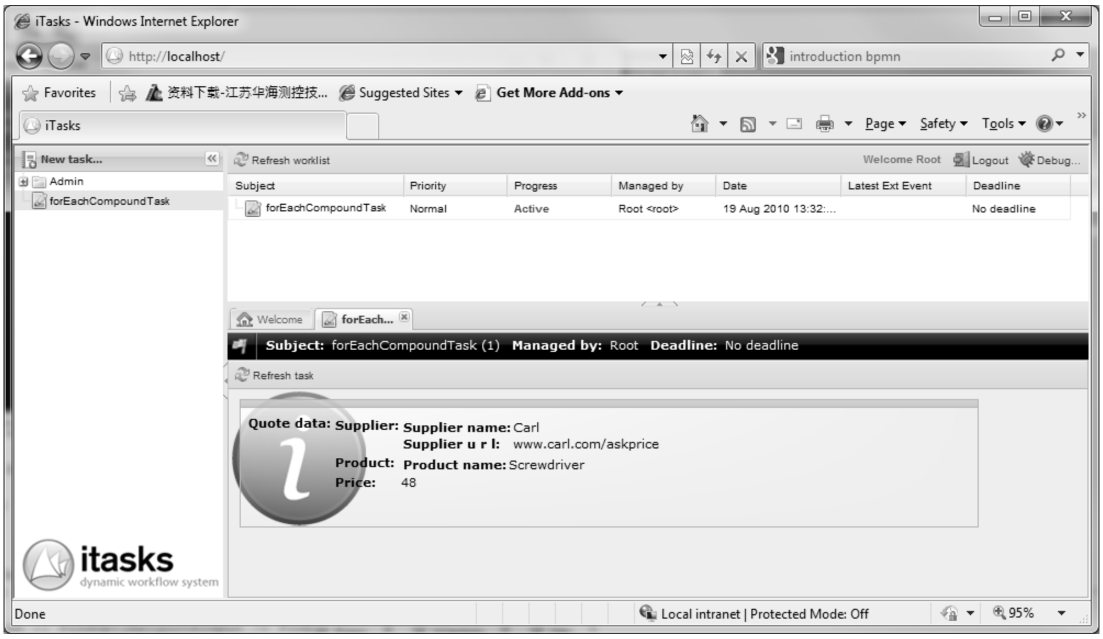
Subject: forEachCompoundTask (1) Managed by: Root Deadline: No deadline

Output from subsystem

Quote data: Supplier: Supplier name: Alice
Supplier u r l: www.alice.com/makequote
Product: Product name: Screwdriver
Price: 56

3: A message box with title 'output from subsystem' and content {QuoteData = {Supplier = { Supplier name = "Alice", Supplier URL = "www.alice.com/makequote"}, ProductData = {"ProductName = "Screwdriver"}, Price = 56 }} is shown

4: Step 2 and 3 are repeated until every element of the list is processed.



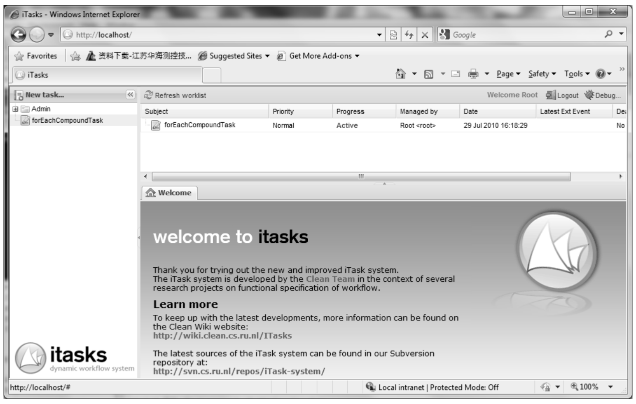
Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline
forEachCompoundTask	Normal	Active	Root <root>	19 Aug 2010 13:32:...		No deadline

Subject: forEachCompoundTask (1) Managed by: Root Deadline: No deadline

Quote data: Supplier: Supplier name: Carl
Supplier u r l: www.carl.com/askprice
Product: Product name: Screwdriver
Price: 48

5: The data {QuoteData = {Supplier = { Supplier name = "Carl", Supplier URL = "www.carl.com/askprice"}, ProductData = {"ProductName = "Screwdriver"}, Price = 48 }} is returned as being the cheapest supplier.

For the scenario without suppliers in the input the following sequence occurs:



Subject	Priority	Progress	Managed by	Date	Latest Ext Event	De
forEachCompoundTask	Normal	Active	Root <root>	29 Jul 2010 16:18:29		No

welcome to itasks

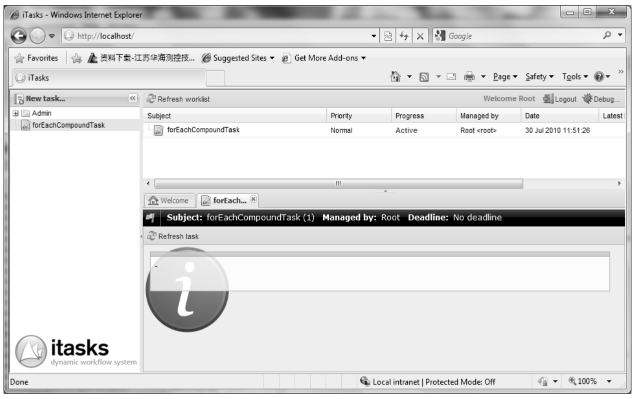
Thank you for trying out the new and improved iTask system.
The iTask system is developed by the Clean Team in the context of several research projects on functional specification of workflow.

Learn more

To keep up with the latest developments, more information can be found on the Clean Wiki website:
http://wiki.clean.cs.ru.nl/ITasks

The latest sources of the iTask system can be found in our Subversion repository at:
http://svn.cs.ru.nl/repos/iTask-system/

1a: The task is initiated



Subject	Priority	Progress	Managed by	Date	Latest
forEachCompoundTask	Normal	Active	Root <root>	30 Jul 2010 11:51:26	

Subject: forEachCompoundTask (1) **Managed by:** Root **Deadline:** No deadline

-

2a: The data – is shown as return value, representing Nothing.

The scenario with suppliers is linked to the use case as follows:

Use case	0	1	2	3	4	5	
Scenario	1	4	2	3	jump	5	

For the scenario without suppliers the alternative path of the use case is accessed:

Use case	0	1a	2a				
Scenario	1a	jump	2a				

Every step in the use case has been covered, which implies a certain level of confidence in the implementation.

7.4 Validating the mapping of pools and lanes

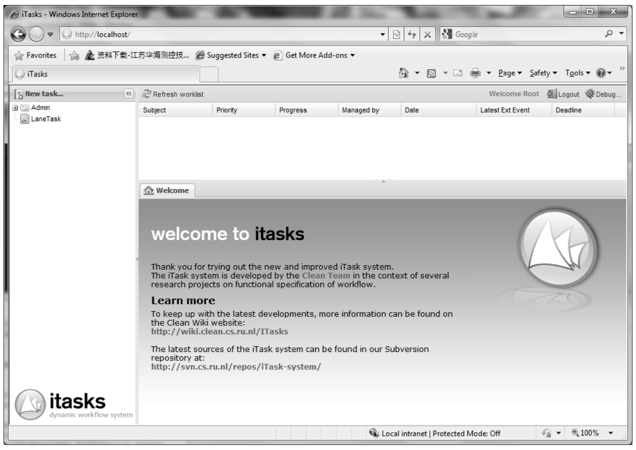
This paragraph will deal with testing the pool and lane implementation. The lane implementation involves three parties that are part of the process, namely 'Webserver', 'Management' and 'Administration. In use case form the following activities are supposed to take place:

Lane activity which involves three actors	
Name	Lane activity which involves three actors
Summary	The lane activities delegates information from a URL to a web server first, which is in return forwarded in processed form to two other parties, namely management and administration
Actors	Webserver, Management, Administration
Basic course of events	0) The activity is triggered 1) System provides URL information to the web server 2) Webserver returns job and credential information to the system 3) The system sends job information to the administrator and credential information to management 4) Management returns a void message back 5) Administrator returns a void message back
Exceptions	
Alternative paths	4a) Administrator returns a void message back 5a) Management returns a void message back

When looking in realtime at the smallest time unit level, it is impossible that both actions of step 3 of the above use case really happen in parallel. However, practically both steps always happen at the same time, which makes it not necessary to provide an order of them. This view matches the statements that were made in the Itasks semantics. However, the return values need to be ordered, because they heavily rely on the reply times of respectively the administrator and management (if they reply at all). In order to make it possible to intercept the activities that are received from and sent to the actors, they are wrapped with an output message (which will be received by user Root as the initiator) around to interface. This gives the following code:

```
DispatchToApprover :: UserId DispatchToApproverInput -> (Task DispatchToApproverOutput)
DispatchToApprover _ input =
  showMessageAbout "Input to Web Server" input >>|
  return (MathematicalCalculation input)
  >>= \output -> showMessageAbout "Output from Web Server" output
  >>| return output

ApproveRequest :: UserId ApproveRequestInput -> (Task Void)
ApproveRequest user input =
  showMessageAbout "Input to Management" input >>|
  user @:("Perform task ApproveRequest please",
  (showStickyMessageAbout "" input) ||- (enterInformation ""))
  >>= \output -> showMessageAbout "Output from Management" output
  >>| return output


PreparePO :: UserId PreparePOInput -> (Task Void)
PreparePO user input =
  showMessageAbout "Input to Administration" input >>|
  user @:("Perform task ApproveRequest please",
  (showStickyMessageAbout "" input) ||- (enterInformation ""))
  >>= \output -> showMessageAbout "Output from Administration" output
  >>| return output
```

Depending on the response times of administrator and management (which are simulated by having the user in the Itasks system press okay) either one of the two scenarios happens. The scenarios are ran from the perspective of the root user and although the root is not mentioned in the above use case, it will be seen as a debugging entity which has an overview and access to what is going on. Until a certain moment there is no difference between the two scenarios and they both go as follows:



Welcome Root

Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline

welcome to itasks

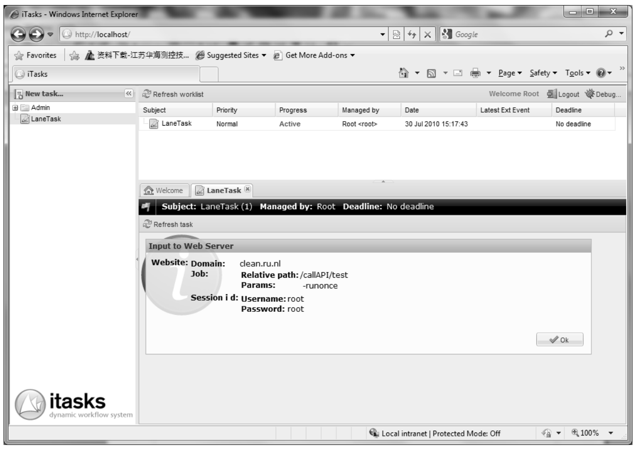
Thank you for trying out the new and improved iTask system.
The iTask system is developed by the Clean Team in the context of several research projects on functional specification of workflow.

Learn more

To keep up with the latest developments, more information can be found on the Clean Wiki website:
http://wiki.clean.cs.ru.nl/ITasks

The latest sources of the iTask system can be found in our Subversion repository at:
http://svn.cs.ru.nl/repos/iTask-system/

1: The task is initiated by double clicking on it.



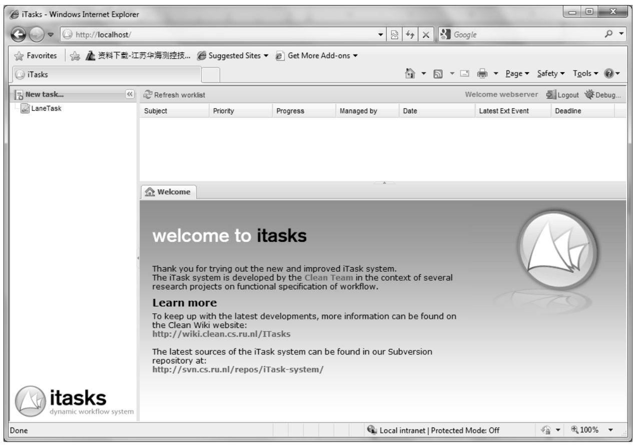
Welcome Root

Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline
LaneTask	Normal	Active	Root <root>	30 Jul 2010 15:17:43		No deadline

Subject: LaneTask (1) Managed by: Root Deadline: No deadline

Input to Web Server
Website: Domain: clean.ru.nl
Job: Relative path: /callAPI/test
Params: -runonce
Session i d: Username: root
Password: root

2: The Root user as an observer can see that information has been send to the webserver



Welcome webserver

Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline

welcome to itasks

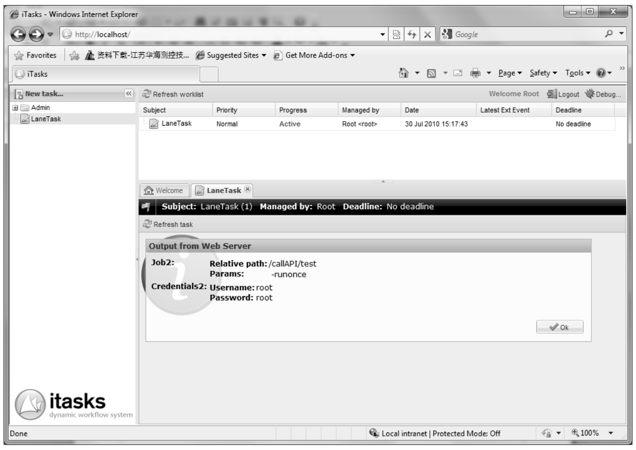
Thank you for trying out the new and improved iTask system.
The iTask system is developed by the Clean Team in the context of several research projects on functional specification of workflow.

Learn more

To keep up with the latest developments, more information can be found on the Clean Wiki website:
http://wiki.clean.cs.ru.nl/ITasks

The latest sources of the iTask system can be found in our Subversion repository at:
http://svn.cs.ru.nl/repos/iTask-system/

3: Logging on as webserver does not provide any information, since this task has been implemented as mathematical function



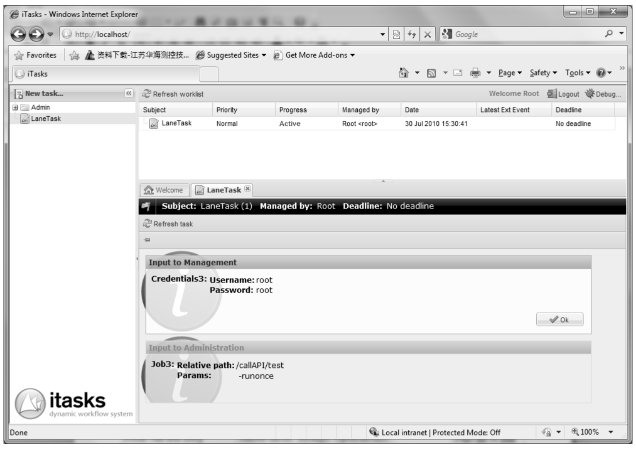
Welcome Root

Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline
LaneTask	Normal	Active	Root <root>	30 Jul 2010 15:17:43		No deadline

Subject: LaneTask (1) Managed by: Root Deadline: No deadline

Output from Web Server
Job2: Relative path: /callAPI/test
Params: -runonce
Credentials2: Username: root
Password: root

4: Clicking okay in the root provides the return call from the webserver, which shows that the URL has been split up in two new entities.



Welcome Root

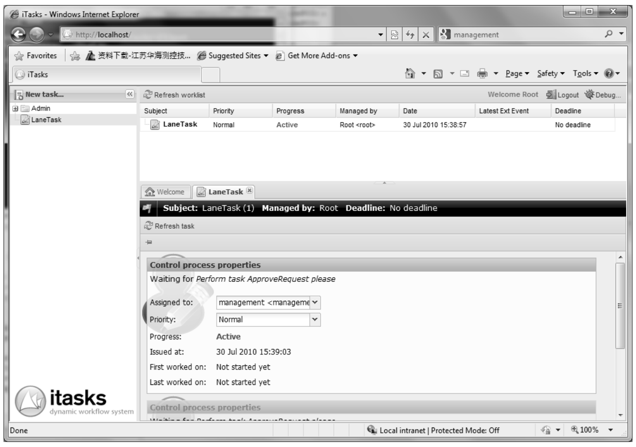
Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline
LaneTask	Normal	Active	Root <root>	30 Jul 2010 15:30:41		No deadline

Subject: LaneTask (1) Managed by: Root Deadline: No deadline

Input to Management
Credentials3: Username: root
Password: root

Input to Administration
Job3: Relative path: /callAPI/test
Params: -runonce

5: Clicking okay in the Root user perspective shows one task with two inputs of respectively the management and administrator task, which are about to start in parallel.



Welcome Root

Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Deadline
LaneTask	Normal	Active	Root <root>	30 Jul 2010 15:38:57		No deadline

Subject: LaneTask (1) Managed by: Root Deadline: No deadline

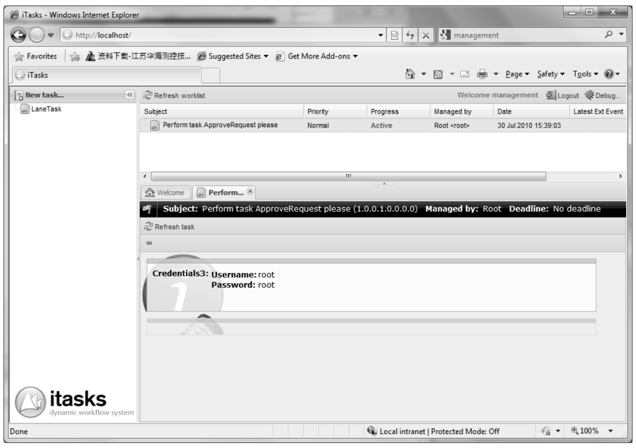
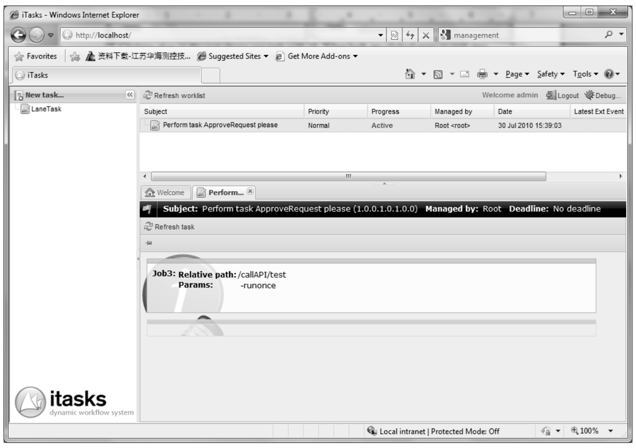
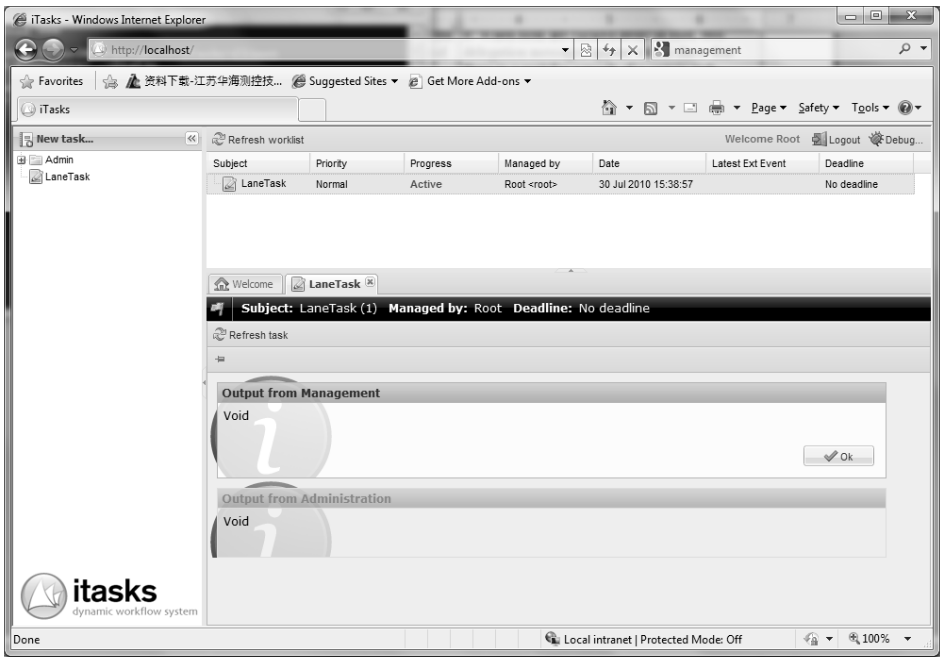
Control process properties
Waiting for *Perform task ApproveRequest please*

Assigned to:	management <manageme
Priority:	Normal
Progress:	Active
Issued at:	30 Jul 2010 15:39:03
First worked on:	Not started yet
Last worked on:	Not started yet

Control process properties

6: When both are clicked away in Root, two delegation messages appear which show that there is a wait for a result of a certain task.

This is the moment when both scenarios start to differ. In one case management finishes its work first (step 7a and 8a), in the other one the administrator (step 7b and 8b):

 Welcome management Subject | Priority | Progress | Managed by | Date | Latest Ext Event Perform task ApproveRequest please | Normal | Active | Root <root> | 30 Jul 2010 15:39:03 **Subject:** Perform task ApproveRequest please (1.0.0.1.0.0.0.0) **Managed by:** Root **Deadline:** No deadline **Credentials3:** **Username:** root **Password:** root	 Welcome admin Subject | Priority | Progress | Managed by | Date | Latest Ext Event Perform task ApproveRequest please | Normal | Active | Root <root> | 30 Jul 2010 15:39:03 **Subject:** Perform task ApproveRequest please (1.0.0.1.0.1.0.0) **Managed by:** Root **Deadline:** No deadline **Job3:** **Relative path:** /callAPI/test **Params:** -runonce
7a: Logging in as management shows there is work to do, namely information {credentials3 = { username = "root", password = "root" }} is shown and a reply of Void (just pressing okay) is expected in return. Clicking returns the message.	7b:Logging in as administrator shows there is work to do, namely information {job3 = {relativePath = "/callAPI/test", params= "-runonce"}} is shown and a reply of Void (just pressing okay) is expected in return. Clicking returns the message.
8a: see 7b	8b: see 7a
 Welcome Root Subject | Priority | Progress | Managed by | Date | Latest Ext Event | Deadline LaneTask | Normal | Active | Root <root> | 30 Jul 2010 15:38:57 | | No deadline **Subject:** LaneTask (1) **Managed by:** Root **Deadline:** No deadline **Output from Management** Void **Output from Administration** Void 9: Two return values of type Void are shown in one task, indicating that both delegations have been successful.	

The scenario in which management finishes first associated with the use case as follows:

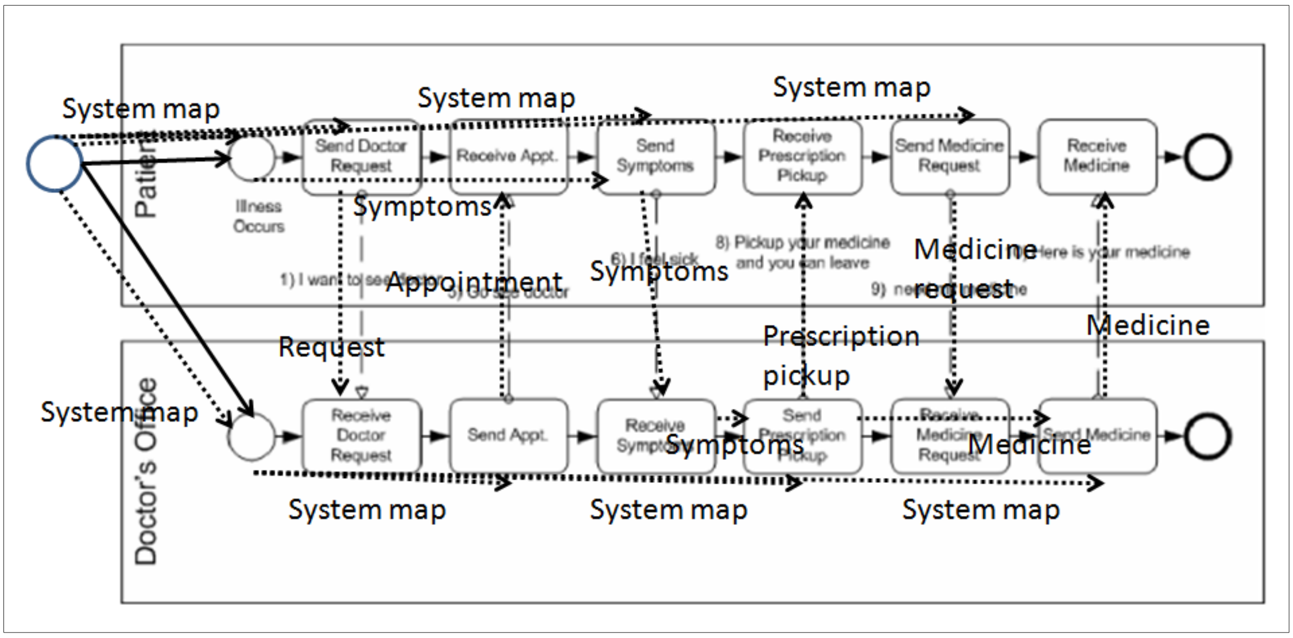
Use case	0	1	2	3	4	5	
Scenario	1	2	4	5	7a	8a	

While the other scenario provides the following table:

Use case	0	1	2	3	4a	5a	
Scenario	1	2	4	5	7b	8b	

Both use cases are covered.

Second the concept of pools is tested, which requires a bit more effort. Recap that the BPMN diagram including data flow is as follows:



When looking at the doctor's office's system only, which will be the area of testing, initially only one flow can be detected. This flow starts with an event, goes to 'Receive Doctor Request', passes several events, eventually reaches 'Send medicine' and ends with an event. When a message is sent to another system, nothing particular happens, just a call to a webservice. When the remote system is ready to receive it, an acknowledgment will return right away and when the remote system is not ready yet, still an acknowledgment is the result (be it somewhat later or not).

The interesting thing with respect to flows happens when a message needs to be received. When this is happening, two cases apply, namely the one in which the message has been sent from the remote system and stored before and the one case in which the message has not. The hard thing about this, is that this message could have been sent anytime before in the use case (eg. ten steps earlier in the basic course of events). This boost the number of possible use case paths enormously. In order to deal with this, two use cases have been defined, of which one deals with the message handling and the other with the doctor's office's system. After this the only distinction that is made in the doctor's office's system for receiving messages, is that it either has been sent or has not. This leads to the following two use cases:

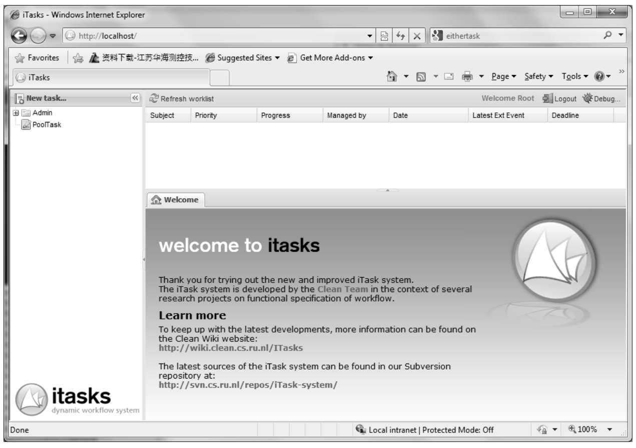
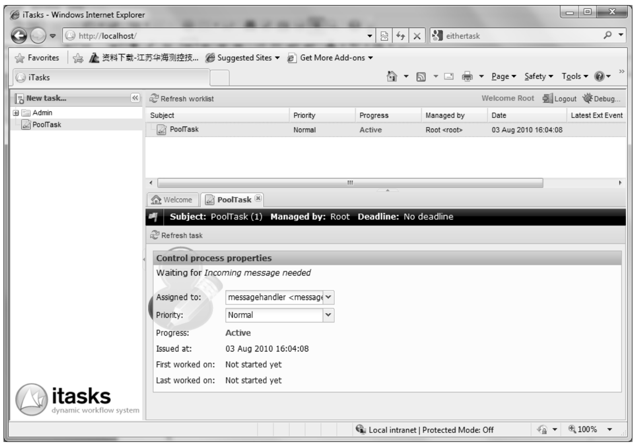
Message handler dealing with incoming message	
Name	Message handler dealing with incoming message
Summary	The local system eventually needs a message from a remote system. In order to make it possible for the local system to receive it, a message handler is used to synchronize both system's activities
Actors	Local system, remote system
Basic course of events	0) The message handler is triggered by a business process that involves more pools and receives information of all parties involved 1) Remote system sends a message and its own identity to the message handler 2) The local system sends a request for receiving a message from the remote system to the message handler 3) The message handler sends the message to the local system
Exceptions	
Alternative paths	1a) The local system send a request for receiving a message from the remote system to the message handler 2a) Remote system sends a message and its own identity to the message handler 3a) The message handler sends the message to the local system

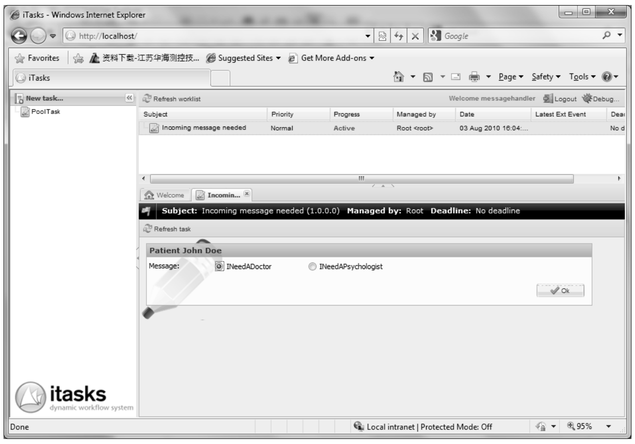
Unfortunately due to a lack of implementation the above use case cannot be tested. The assumption however is that the message handler is functioning perfectly according to the above specification. As the scenario later on will show, its functioning is simulated by an extra Itasks user on the system.

The use case really implementing the pools is as follows:

A doctor's office dealing with a patient as an external system	
Name	A doctor's office dealing with a patient as an external system
Summary	A doctor's office deals with a patient whenever symptoms of a disease occur. The patient does not have access to the Itasks system and might have his own business process and system. Communications happen through messages, which occur at fixed points in the business process
Actors	Patient, Doctor, Message handler
Basic course of events	0) Patient has an illness and triggers through the definition of pools a new instance of the doctor's office's process 1) System sends request for next message to the message handler 2) Message handler sends request to see doctor (from patient) to the system 3) System sends appointment time to the patient 4) System sends appointment time to the doctor 5) System sends request for next message to the message handler 6) Message handler sends symptoms (from patient) to the system 7) System sends a prescription pickup request to the patient 8) System sends request for next message to the message handler 9) Message handler sends medicine request (from patient) to the system 10) System sends the medicine to the patient
Exceptions	
Alternative paths	

Note that the introduction of the message handler use case made the second use case much simpler. This is due to the fact that the message handler deals with the 2 different scenarios that occur (receiver reaches synchronization point first versus sender reaches synchronization point first) and therefore creates and abstraction for the doctor's office about when the message has actually been sent by the patient. When the implementation from the previous chapter is executed, the following scenario takes place:

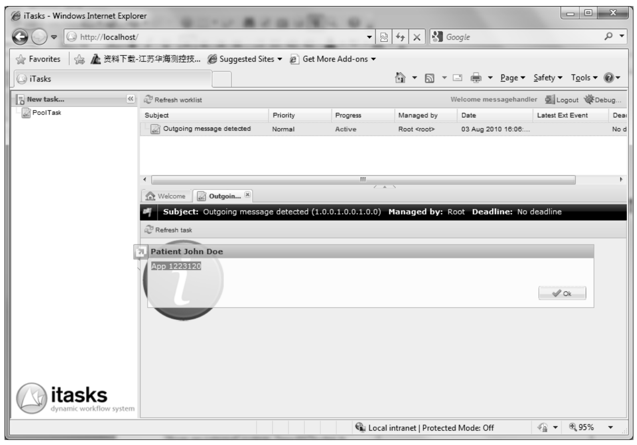
 welcome to itasks Thank you for trying out the new and improved iTask system. The iTask system is developed by the Clean Team in the context of several research projects on functional specification of workflow. Learn more To keep up with the latest developments, more information can be found on the Clean Wiki website: http://wiki.clean.cs.ru.nl/ITasks The latest sources of the iTask system can be found in our Subversion repository at: http://svn.cs.ru.nl/repos/iTask-system/	 Subject: PoolTask (1) Managed by: Root Deadline: No deadline Control process properties Waiting for *Incoming message needed* Assigned to: messagehandler <message Priority: Normal Progress: Active Issued at: 03 Aug 2010 16:04:08 First worked on: Not started yet Last worked on: Not started yet
1: The task is initiated as Root user by a double click.	2: A task has been assigned to the message handler.


Subject: Incoming message needed (1.0.0.0) Managed by: Root Deadline: No deadline

Patient John Doe

Message: INeedADoctor / INeedAPsychologist

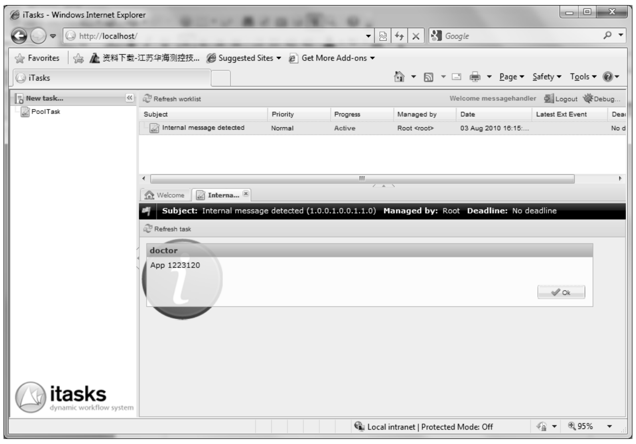
3: Logging in as messagehandler shows that an incoming message is needed, which simulates input from an external system. IneedADoctor is chosen.


Subject: Outgoing message detected (1.0.0.1.0.0.1.0.0) Managed by: Root Deadline: No deadline

Patient John Doe

App 1223120

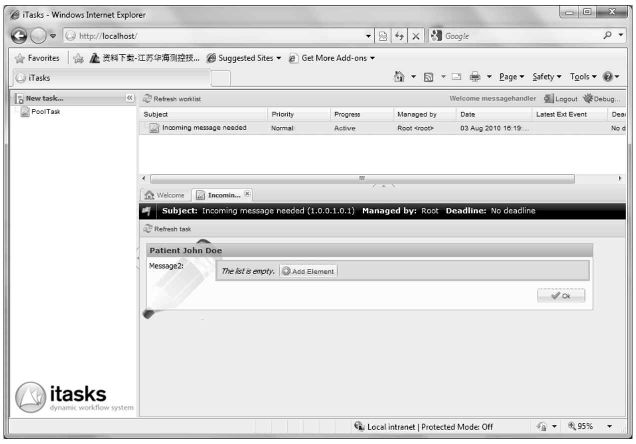
4: An outgoing message to patient 'John Doe' is detected, containing 'App 1223120', representing an appointment at December 23rd, 12:00. Okay is pressed.


Subject: Internal message detected (1.0.0.1.0.0.1.1.0) Managed by: Root Deadline: No deadline

doctor

App 1223120

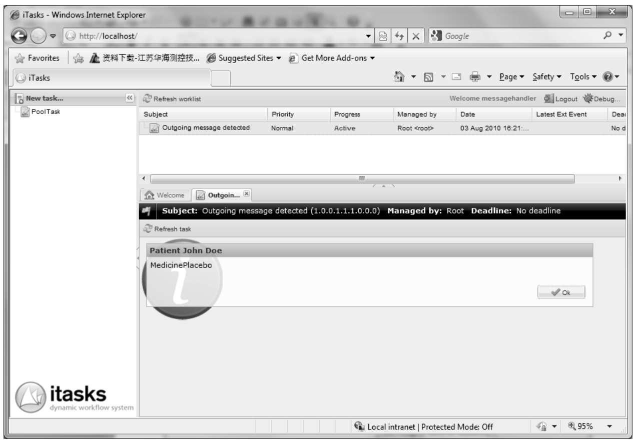
5: An internal message to the doctor is detected, containing 'App 1223120'. Okay is pressed.


Subject: Incoming message needed (1.0.0.1.0.0.1) Managed by: Root Deadline: No deadline

Patient John Doe

Message2: The list is empty. Add Element

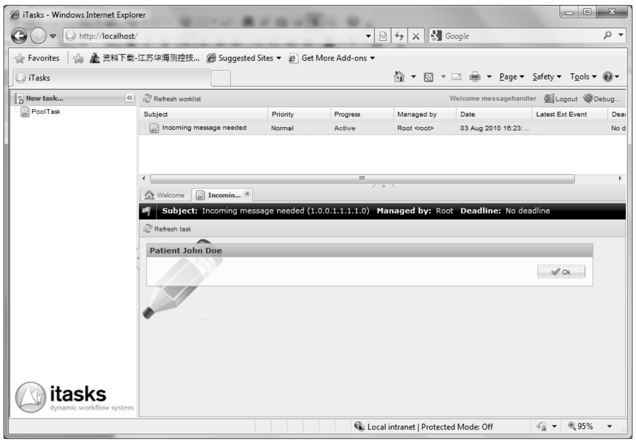
6: An incoming message is needed from the patient, which contains his symptoms. Nothing (the empty list) is entered in the messagehandler.


Subject: Outgoing message detected (1.0.0.1.1.1.0.0.0) Managed by: Root Deadline: No deadline

Patient John Doe

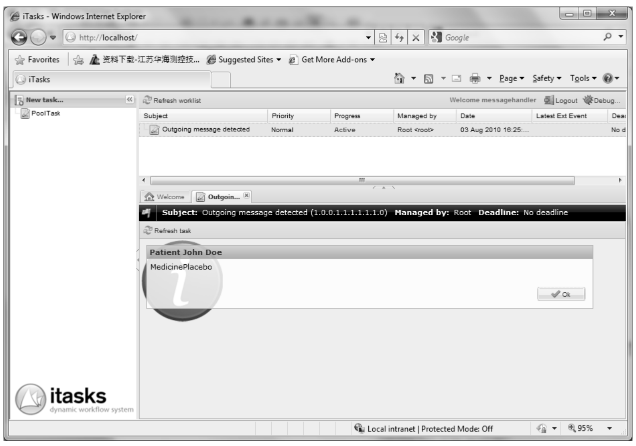
MedicinePlacebo

7: An outgoing message to the patient containing 'MedicinePlacebo' (normally a great cure for patients without symptoms) is sent, representing a prescription the patient can pick up.


Subject: Incoming message needed (1.0.0.1.1.1.1.0) Managed by: Root Deadline: No deadline

Patient John Doe

8: An incoming empty message from John is needed, representing his request for medicine. Okay is pressed in the messagehandler.

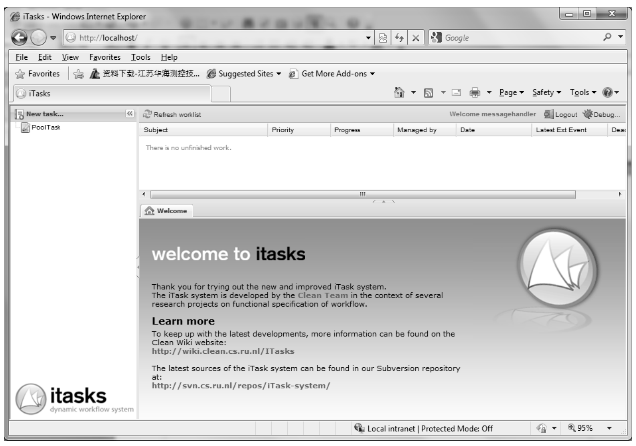


Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Dea
Outgoing message detected	Normal	Active	Root <root>	03 Aug 2010 16:25:...		No d

Subject: Outgoing message detected (1.0.0.1.1.1.1.1.1.0) **Managed by:** Root **Deadline:** No deadline

Patient John Doe
MedicinePlacebo

9: An outgoing message to the patient containing 'MedicinePlacebo' is sent, representing the actually handing over of the medicine.



Subject	Priority	Progress	Managed by	Date	Latest Ext Event	Dea

There is no unfinished work.

welcome to **itasks**

Thank you for trying out the new and improved iTask system.
The iTask system is developed by the Clean Team in the context of several research projects on functional specification of workflow.

Learn more

To keep up with the latest developments, more information can be found on the Clean Wiki website:
http://wiki.clean.cs.ru.nl/ITasks

The latest sources of the iTask system can be found in our Subversion repository at:
http://svn.cs.ru.nl/repos/iTask-system/

10: Going back to the root user shows that the whole process has finished.

Step 3, 6 and 8 deal with an incoming message and for those steps the use case about the message handler thus applies. For each of them 2 scenarios may occur, which are as follows:

– The sender from external system reaches the messaging point first: In this case the message is stored in something like a mailbox. Whenever the point is reached at which the data is needed, it is simply retrieved from the mailbox and 'entered' in the form of the message handler. The is the regular BCOE of the use case (step 0,1,2 and 3).
– The receiver of local system reaches messaging point first: In this case a prompt for the message occurs. This is similar as waiting for an input from a user as in an enterInformation task. Since enterInformation is used to symbolize this, no mismatch can occur and thus is the alternative path of the use case (step 0,1,2a and 3a) covered. Note that in this case the message does not need to be stored (or merely for buffering purposes).

The just mentioned makes it assumable that the message handler is doing its job right (or can be designed to do so). What is left is to match the steps of the scenario with the use case and show that all use case steps are covered. This leads to the following links:

Use case	0	1	2	3	4	5	6
Scenario	1	2	3	4	5	As 2	6

Use case	7	8	9	10			
Scenario	7	As 2	8	9			

Step 5 and 8 of the use case have been omitted, because the occur exactly in the same way as step 2 of the scenario. This avoids meaningless redundancy in the scenario. From this can be concluded that all steps of the use case have been covered appropriately.

7.5 Conclusion

This chapter has taken the generated Clean code from the previous chapter as a starting point. Use case diagrams have been generated from the example BPMN diagrams first, after which the Clean code is executed. The behavior of the code has been compared with what is supposed to happen in the use cases and a match has been found between them. Be aware however, that the generation of the use cases and the comparison are very subjective and that no scientific conclusions can be addressed to this work. A good step to take here would be a mathematical proof, or finding consensus among a group of people to reach the intersubjective agreement stage.

8 Conclusion

8.1 Achievements and problems: A successful mapping between BPMN and Itasks has been made

Several steps have been taken in order to answer the research question, which is: How can the BPMN meta model be expressed in terms of a functional language? First a literature study was conducted in which different concepts have found to be important, namely workflows, BPMN and functional approaches. This led to the choice to formalize BPMN and create a proof of concept implementation for it in the Itasks library for the Clean language.

The formalization of the BPMN language was done in ORM and led to a meta model in which the concepts and their relationships were described mathematically. For example BPMN diagrams a population in terms of the meta model has been given. This formalization has been done by looking at example BPMN diagrams, checking the specification and using common sense about what their meaning is. This is the right way to go since BPMN is supposed to rely a lot on intuition for its meaning. The BPMN diagram shows objects at a certain position and size, but for the implementation this position and size have chosen to be irrelevant. However, if one wants to reproduce the original BPMN diagram, this information is needed. This information has been introduced as an addition to the meta model by adding position and dimension data.

After this, it turned out that the BPMN diagram so far did not have a meaning implementation wise. First of all a data layer was added to it in the meta model, to show how data flows through different activities. After this four different techniques have been introduced to implement an atomic activity, the smallest building brick of a BPMN diagram. They are a user prompt, calling a web service, accessing a data source and doing a mathematical calculation.

Using the example diagrams and adding some fictional meaning to them, led to an implementation in the Clean language using the Itasks library. This translation has been done in an as systematic way as possible in order to make it assumable that this could be automated in the future. The result of this work contains several small code examples, each one describing a certain concept of the BPMN language. To make it assumable that the translation has been done systematically, a pseudo algorithm describing it has been given as well.

In order to have some confidence that the implementation is doing what it is supposed to do, use cases have been manually generated from the BPMN examples. After this the software has been ran, and a check has been made to see whether the scenarios the software caused matched the corresponding use case. In all cases this turned out to be the case.

For scientific relevance this paper has made it possible to reason about BPMN constructs in advanced ways (higher order, strongly typed, lazy evaluation) and see how this relates to the execution of an implementation. For the Itasks community the paper has confirmed that many BPMN constructs are doable with their libraries and what to do with their language to appeal the business people. The missing aspects were the support for standards (SOAP/WSDL), the inability to synchronize with other systems in case of messaging and the lacking functionality/documentation of the API, which all made it hard to connect a Itasks system to an external system (which in return could be another Itasks system).

For social relevance the thesis has contributed to be able to translate a BPMN diagram rapidly to a working GUI, which makes it possible to show to a business person in prototype form what the meaning of a certain diagram is. In addition to this, this thesis might inspire software producers to add certain concepts to their BPMN based software which makes the software more efficient or higher quality.

8.2 Future work

Although a lot of work has been done to formalize BPMN in a functional way in this thesis, many topics have remained uncovered. In addition to this this thesis has revealed a lot of more work that can be done in the future. This is covered in this paragraph using five subparagraphs.

8.2.1 Changes and addition that could be made to the meta model

Despite the offering of a quite extensive meta model in this thesis, there is always room for improvement. First of all this meta model has been built by one single person and certain design decisions have been made, of which some are a matter of taste. It would be interesting to use the meta model as a starting point of discussion and have different people form different disciplines look at it and come to some consensus about what way to go, regarding the design decisions. Ideally this would lead to a standard everyone is willing to comply to.
The current meta model does not include the instance level of processes. So far this has not been necessary, because the Itasks system provides this functionality. However, if one wants to have a more complete picture about the system and reason about it theoretically, it is very recommended to include this part in the meta model. This provides opportunities to check whether what the Itasks system is doing is scientifically justifiable, but also gives opportunities to have constraints on the instance level and to reason about scheduling, which will be noted more elaborately later on.

In addition to the instance level, the resource level is something that could be added as well. In the Itasks system, resources are represented by users and roles they can attend in. The roles represent some sort of capabilities they have, which are decisive in what tasks they can and cannot perform. This information can be once again used for scheduling, by assigning certain tasks to certain resources in a certain order.

Since the Itasks and Clean programming environment have been used for implementation quite a lot (especially in the mathematical function implementation), it might be useful to investigate how this environment could be added to the meta model. Of course one could choose to just copy and paste Clean code to the meta model population. However, when a deeper understanding is needed, it requires that this code is parsed and semantically explained first, leading to more entities and concepts in the meta model.

The current meta model as presented in this thesis, was shown in the ORM language, but did not or hardly made use of labels. Labels are a way in ORM to state how entities are actually stored physically on a computer. The not using of labels has purposely been done, since the representation of the concepts does not really change what is really going on. By delaying the choice the actual implementation is left open. Implicitly some representation has been chosen occasionally, in order to transform the population to Clean code. However, in order to really store the model and compute with it on a computer, decisions have to be made with respect to the labels more explicitly. After this has been done, a relational database model or XML representation of the meta model and population can be given, which makes it possible to store everything that is going on. But this data can also be used to have communication about it in a formal way between different pieces of software.

The meta model that has been created has mostly been based on BPMN version 1. At the time of writing BPMN is developing further and the meta model that has been presented here, might need some adjustments or additions to cope with it. What also turned out before, and can be seen from Appendix C, is that BPMN offers many gateways and events which have been generalized in the current meta model. It might be possible to assign a more specific meaning to them and have this translated to certain constructions at the implementation level. One interesting concept here that still needs to be covered, is that an instance of a BPMN process, can trigger another instance.

In the literature study of this thesis, the language BPEL was noted and rejected because of it technical form which is hard to understand for non-IT people. For reasoning purposes and scientific purposes however, it might be valuable to see how the implementation as presented in this thesis would look like in BPEL and to see if a link between BPEL and Itasks constructs can be made. This in order to make Itasks more accessible to the non-functional programming people and also to see how supportive Itasks is with respect to what it is aiming to achieve.

Finally it would be interesting to see how the meta model and the solutions connect to the concrete commercial and open source solutions that are available on the market already. Options for comparison are Cordys' Business Operating Platform[R32], Microsoft's Biztalk[R33], Oracle Business Process Management[R34], Appian[R35] and IBM's BPM suite[R36].

8.2.2 Changes and addition with respect to Itasks and the implementation of BPMN

The Itasks system has been used in this thesis to take care of the implementation level. Since Itasks has not been tested on whether or not supporting BPMN before, it was very likely to assume that some constructs that were needed are lacking in the system. This turned out to be the case. It is worth noting that Itasks does support most of the BPMN constructs though, which has been shown by the implementation examples in this thesis.

There are two aspects at which the Itasks system might be improved with respect to BPMN support, but also in general. First of all the API does not support the injection of information in a task. In addition to this the API is not yet well documented and might need some examples about how to use it in different programming languages (Eg. Creating a workflow in JAVA and dealing with the result). The second aspects deal with complying to standards. The RPC daemon deals with a REST call and returns something in JSON format, which is considered good. However, in addition to this there is a big trend to support SOAP/WSDL webservices on the market. This can be implemented for the API, as well as for making a call in a task to the outside world (ideally by having a special data type for webservices and still using enterInformation and displayMessage). Although this might not directly provide scientific value and seems to cause overhead in making calls, it will make it eventually easier to connect Itasks to new systems and to expect people from outside the Itasks community to do something with the system. Other ways in which standards could be applied involve the usage of the data type definitions from geoJSON[R14], Google API[R21][R22] and Microsoft Bing API[R23].

This thesis has provided a proof of concept in Clean code about how BPMN constructs could be implemented. This involved a lot of small and 'quick and dirty' code fragments to illustrate what has been going on and to make it very assumable that this code could be automatically generated. When one wants to use the BPMN constructs occasionally in the Itasks system, this code could be more generalized and changed a bit to make it more usable and extensible in different contexts. This would require the use of the following properties among others:

- Polymorphism: The functions that are made for BPMN and BPM should be able to deal with different types. (eg. At least the types for which certain functions such as gPrint have been derived).
- Libraries: All the functionality should be packed in a easy distributable package (in Clean named module) named BPMN and examples and documentation about how to use it should be added.
- Generics: When everything is set up in a very general way, it becomes possible to reason about BPMN at a generic level and have certain actions take place independent of the data type (Eg. A BPMN diagram could be described by a data type after which generic functions make it possible to deal and reason with it).

So far the testing of the implementation has been done through the generating of use cases. When the BPMN diagrams are applied on real life cases, the testing becomes more sophisticated and the confidence one can have in the result increases. Therefore finding a real life case to apply the theory on might be a good option. In addition to this a more formal way of proving is desirable. Creating the use cases has been an informal process, relying a bit on intuition and also linking the use cases to the scenarios is not completely formal. A way to solve this is to write the Clean code at a meta level, which has been discussed just before. In addition to this semantic proof trees with induction and deduction rules, which are based on the semantic rules of Clean could be created.

Also when one goes the other direction, namely from Itasks to BPMN, possibilities arise. What has become clear when investigating the Itasks system and the BPMN language is that in general the Itasks system is much more powerful and expressive. It might be the case that some constructs from the Itasks and Clean language are very useful for a business context while they are not (yet) expressible in or part of the BPMN language. When one wants to extend the BPMN language with Itasks and Clean constructs one has to keep the following properties in mind:

- Simplicity: The BPMN notation is made to be simple and to be understood by people who do not even have a formal background. When one adds a construct from the Itasks language one needs to be sure that the simplicity property is kept alive.
- Notatability: The BPMN language is mostly based on graphical constructs which improves seeing what is going on for a human. When an Itasks construction is added to the extension of the BPMN language, one needs to be sure that it possible to express it graphically in a way that intuitively makes sense. This needs to be the case independent of the value it can have for the business in functional sense.
- Usability: One should not aim to add an Itasks construct to the BPMN language extension, just to show that something technologically is possible. Instead of ICT, the focus should be mainly on people and organizations, since that is what BPMN is trying to improve.

Examples of constructs which might be useful and not yet part of the BPMN language are:

- Recursion: Doing a certain task encompasses doing smaller similar tasks, such as the assembling of a big product requires the assembling of its smaller parts first.

- Polymorphism: Doing a certain task for different entities is done in a similar way. For example, handling an order at a hamburger restaurant is done doing certain steps (asking order, packing products, paying order), no matter what someone orders.
- Infinite data structures: Sometimes it might be conceptually useful to deal with infinite data structures. As long as these structures do not need to be evaluated completely, this is not a problem. In a business context one could for example at the sales department assume that stocks are infinitely big as long as procurement can keep up with them.
- Higher order concepts: A complete paragraph (combined with scheduling) is dedicated to this topic later on.

8.2.3 The creation of a modeling tool for executable BPMN

Basically everything that happened in this thesis, happened manually. One chapter was dedicated to being able to graphically reproduce the original BPMN diagram. This calls for a tool in which the modeling of BPMN can be done automatically. Since BPMN is commonly used all over the globe[R24], a lot of software to make models is available already[R25][R26][R27][R28]. The output of the software could be used to populate the meta model as presented in this thesis, but one could also decide to write a completely new modeling tool that deals with everything as presented in this thesis. Ideally the tool would even be an Itasks task of type BPMN and lead to a BPMN editor whenever the enterInformation task for the type BPMN is used and display a BPMN diagram when showInformation is accessed.

8.2.4 Adding scheduling and higher order concepts to the BPMN meta model and implementation

This subparagraph deals with building an extra layer on the current Itasks system in order to be able to support more constructs that are wanted from the management's perspective. They can be made by extending the Itasks system at the lowest level, but also by introducing new combinators and constructs in the Clean language. A combination of both methods is an option as well.

One topic which plays are role here and has briefly been mentioned already is scheduling. In the current Itasks system, one just triggers a new workflow whenever one feels like and a whole process is executed. There is support for delegation using the @: combinator, but not yet the possibility to have something scheduled automatically. The scheduling algorithm, which assigns tasks to different users in a certain order, has to keep the following properties among others in mind:

- Occupation of resources: Depending on what has been assigned to the resources already and their general availability (eg someone might take a vacation day), the scheduler can or cannot assign tasks to a certain resource.
- Expected task durations: The scheduler needs to know how long a task normally takes in order to "predict" the future, which is a common property of scheduling.
- Deadlines of tasks: If a tasks needs to be done soon, it might need higher priority and even though a task got available later, it might be put in front of a task which has been waiting a longer time.
- Non-determinism: Unexpected things might happen (eg a resource becoming unavailable or a task taking longer then planned) on which the scheduler ideally responds in real time by rescheduling.

The reason why the just mentioned scheduling problems are possible to implement directly in the Itasks system, is partly due to the support of higher order functions in Clean. This makes it possible to reason about tasks on a meta level and use them as parameters of a function.

Other ways the higher order functions could be used are:

- Applying management theories: Management theories are often about controlling what is going on the organization. When one can reason at a higher level (which higher order functions offer), it becomes possible to detect what is going on and see how it fits in what the organization wants to happen. Data can be used for managers to make decisions, but also to take corrective actions to stay on the right track. The management can be done by a human entity, but also be automated and only involve human interference when complex questions need to be answered.
- Reasoning about the meta model: When the transformation from BPMN to a working implementation is done by defining at a meta level what the meaning of the BPMN constructs is as described before, it becomes possible to feed this implementation to functions that reasons about the meta model (either with or without a population). This can make it possible to have new insights about the meta model, but also to prove properties one want to be true in any case.

8.2.5 Looking back at the literature study

In the beginning of this thesis a literature study has been conducted, leading to many findings. Some of these findings have been used (either explicitly or implicitely), while others have been remained unused. Ideally the result of this thesis could be embedded in the literature again or it should be possible to recognize concepts of the literature in the final result. Since this feedback loop has not been part of this thesis, this subparagraph is at least at aiming what could be done with the result of this thesis with respect to the literature.

First of all Van der Aalst's three dimensions at described in his paper[3]. Even though the resource and case dimension have not been completely covered as found out before, it should be able to detect the different dimensions in the meta model and categorize the objects and roles. This makes it possible to have a certain view on the meta model, which might lead to new insights. Also it will illustrate where to add the resource and case dimension to the model.

Second De Sitter's paper about organizational design[2] which is about how to divide the work. This paper could contribute to the scheduling problem, because it is about how to divide work, but even more importantly it is about how to split up one big chuck of work at all. This includes creating a business process, but also recognizing different milestones and seeing how they can be overseen by management. This theory could be applied on the meta model and one could investigate the consequences (eg. perhaps extra constraints are needed on the meta model).

Third the alternatives to the BPMN language. Even though BPMN is widely supported[R24], it is worth to look around. UML, BEMN, Petri nets and CEP have been mentioned as alternative modeling languages. Since a meta model for BPMN is available now, it might be possible to see how these other languages would fit in this model and whether (partial) transformations are possible. Also it might be possible to extend the meta model in order to make it compatible with different languages at the same time and connect them.

Fourth the functional paradigm that has been chosen. It is possible to see whether other paradigms (such as object oriented, aspect oriented, imperative) are suitable for representing BPMN constructs and what they would look like. The decision can be taken independent of whether uses the meta model as proposed in this thesis.

9 References

References that have been mentioned in this paper come in two flavors, namely:

- Research papers and books: Links to scientific papers and book, which have been marked with a number.
- Online resources: URLs on the internet that provide extra information or the option to download something, which have been marked with an R followed by a number.

9.1 Research papers and books

[1] WR Ashby, 1962, Principles of Self-organization, E:CO Special Double Issue Vol. 6 Nos. 1-2 2004 pp. 102-126

[2] L. Ulbo De Sitter, J. Friso Den Hertog and Ben Dankbaar, 1997, From Complex Organizations with Simple Jobs to Simple Organizations with Complex Jobs, Volume 50, Number 5, 497-534, DOI: 10.1023/A:1016987702271

[3] Wil M. P. van der Aalst, 1999, Process-oriented architectures for electronic commerce and interorganizational workflow, Information Systems Volume 24, Issue 8, December 1999, Pages 639-671

[4] FGB van den Berg, 2008, Creating a meta model describing a production system's concepts and their relations, Bachelor thesis Information Science, Radboud University Nijmegen

[5] Rinus Plasmeijer,Peter Achten and Pieter Koopman, 2008, An Introduction to iTasks: Defining Interactive WorkFlows for the Web, CEFP2007,LNCS5161,pp.1–40

[6] Terje Wahl, Guttorm Sindre, 2006, An Analytical Evaluation of BPMN Using a Semiotic Quality Framework, Advanced Topics in Database Research, Volume 5, 94-105 pp.

[7] SA White, Introduction to BPMNbptrends.com, 2004 IBM Cooperation

[8] Cordys, 2009, Cordys Business Process Management Suite, ROI Realizing - cordys.com Cordys BPM_whitepaper_23062009

[9] Rinus Plasmeijer,Peter Achten and Pieter Koopman, 2007, iTasks: executable specifications of interactive work flow systems for the web, Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, Pages: 141 - 152

[10] Niels Lohmann, 2007, A feature-complete Petri net semantics for WS-BPEL 2.0, Proceedings of the 4th international conference on Web services and formal methods

[11] Rinus Plasmeijer, 2008, Peter Achten and Pieter Koopman, An Executable and Testable Semantics for iTasks, IFL 2008

[12] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer and M. J. Plasmeijer, 1987, Clean — A language for functional graph rewriting, Lecture Notes in Computer Science, 1987, Volume 274, Functional Programming Languages and Computer Architecture, Pages 364-384

[13] W. M. P. van der Aalst, 1998, The Application of Petri Nets to Workflow Management, The Journal of Circuits, Systems and Computers, Vol. 8, No. 1. (1998), pp. 21-66.

[14] Jan Martin Jansen, 2010, Embedding a Web-Based Workflow Management System in a Functional Language, Faculty of Military Sciences

[15] JM Jansen, P Koopman, R Plasmeijer, 2007, Efficient Interpretation by Transforming Data Types and Patterns to Functions, Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, 19-21

[16] SA White, 2005, Using BPMN to model a BPEL process, bptrends.com

[17] C Ouyang, M Dumas, 2006, Translating bpmn to bpel, Citeseer

[18] Recker, Jan and Mendling, Jan, 2006, On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In Latour, Thibaud and Petit, Michael, Eds. Proceedings 18th International Conference on Advanced Information Systems Engineering. Proceedings of Workshops and Doctoral Consortiums, pages pp. 521-532.

[19] Davide Prandi, Paola Quaglia and Nicola Zannone, 2008, Formal Analysis of BPMN Via a Translation into COWS, Lecture Notes in Computer Science, 2008, Volume 5052/2008, 249-263, DOI: 10.1007/978-3-540-68265-3_16

[20] R Dijkman, P Van Gorp, 2010, BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules, is.tm.tue.nl

[21] CR Littler, 1978, Understanding taylorism, British Journal of Sociology

[22] RW Zmud, 1982, Diffusion of modern software practices: influence of centralization and formalization, Management Science

[23] JL Pierce, AL Delbecq, 1977, Organization structure, individual attitudes and innovation, Academy of Management Review

[24] DA Morand, 1995, The role of behavioral formality and informality in the enactment of bureaucratic versus organic organizations, Academy of Management Review

[25] J Kuykendall, 2006, Mapping Police Organizational Change: From a Mechanistic Toward an Organic Model, Volume 20, Issue 2, pages 241–256, August 1982

[26] K Czarnecki, S Helsen, 2003, Classification of model transformation approaches, Techniques in the Context of the Model Driven

[27] WMP Van der Aalst, 2000, Loosely coupled interorganizational workflows: modeling and analyzing workflows crossing organizational boundaries, Information and Management

[28] P Wohed, M Dumas, N Russell, 2006, On the suitability of BPMN for business process modelling, Business Process

[29] C Ou-Yang, 2008, BPMN-based business process model feasibility analysis: a petri net approach, International Journal of Production Research, Volume 46, Issue 14 July 2008 , pages 3763 - 3781

[30] L Bird, A Goodchild, T Halpin, 2000, Object role modelling and xml-schema, Conceptual Modeling—ER 2000

[31] T Halpin, 2006, Object-role modeling (ORM/NIAM), Handbook on Architectures of Information Systems

[32] T Halpin, 1998, Object role modeling: An overview, white paper,(online at www. orm. net)

[33] P Hudak, 19982, Report on the programming language Haskell: a non-strict, purely functional language version 1.2, ACM SIGPLAN Notices Volume 27 , Issue 5

[34] S Thompson, 1999, The Craft of Functional Programming, Citeseer

[35] M Mauny, 1995, Functional programming using Caml Light, Citeseer

[36] G Cousineau, M Mauny, 1998, The functional approach to programming, books.google.com

[37] DA Turner, 1985, Miranda: A non-strict functional language with polymorphic types, Lecture Notes in Computer Science, 1985, Volume 201/1985, 1-16, DOI: 10.1007/3-540-15975-4_26

[38] Tim Bray, Jean Paoli, C. M.Sperberg-McQueen, Tim Bray, 1997, Extensible Markup Language (XML) - Version 1.0 (1997), Citeseer

[39] CF Goldfarb, P Prescod, 2000, XML handbook, ISBN:013055068X

[40] R Akkiraju, 2005, Web service semantics-WSDL-S, Citeseer

[41] R Chinnici, 2007, Web Services Description Language (WSDL) Version 2.0 Part 1: Core Languagew3.org, nelson.w3.org

[42] M Gudgin, M Hadley, N Mendelsohn, 2001, SOAP Version 1.2 Part 1: Messaging Framework, W3C Working Draft 17 December 2001

[43] RP Call, W Fornaciari, 2001, Remote Procedure Call, elet.polimi.it

[44] Brian N. Bershad, 1990, Lightweight remote procedure call, ACM Transactions on Computer Systems (TOCS) Volume 8, Issue 1

[45] Uc Berkeley, 2007, Putting Things to REST, Citeseer

[46] Xu Yang and Zhanhong Xin, 2008, A Study on the Integration Model of EIS Based on SOA, IFIP International Federation for Information Processing, 2008, Volume 254/2008, 627-633, DOI: 10.1007/978-0-387-75902-9_69

[47] HE Eriksson, M Penker, 2000, Business modeling with UML

[48] Decker, G., Grosskopf, A., Barros, A., 2007, A Graphical Notation for Modeling Complex Events in Business Processes, Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International

[49] Murata, T., 1989, Petri nets: Properties, analysis and applications, Proceedings of the IEEE, 541-580

[50] David Luckham, 2008, The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Lecture Notes in Computer Science, 2008, Volume 5321/2008, 3, DOI: 10.1007/978-3-540-88808-6_2

[51] E Proper, 2007, Grounded Enterprise Modelling Version for 2007, Lecture notes

[52] Michael Lawley and Jim Steel, 2006, Practical Declarative Model Transformation with Tefkat, Lecture Notes in Computer Science, 2006, Volume 3844/2006, 139-150, DOI: 10.1007/11663430_15

[53] T Halpin, H Proper, 1995, Database schema transformation and optimization, Lecture Notes in Computer Science, 1995, Volume 1021/1995, 191-203, DOI: 10.1007/BFb0020532

[54] J McCarthy, 1965, LISP 1.5 programmer's manual, books.google.com

[55] D Weinreb, D Moon, 1981, The lisp machine manual, ACM SIGART Bulletin

[56] HP Barendregt, 1984, The lambda calculus: its syntax and semantics, books.google.com

[57] JM Jansen, P Koopman, R Plasmeijer, SB Scholz, 2009, iEditors: extending iTask with interactive plug-ins, Citeseer

[58] M Bravenboer, KT Kalleberg, R Vermaas, 2008, Stratego/XT 0.17. A language and toolset for program transformation, Science of Computer Programming Volume 72, Issues 1-2, 1 June 2008, Pages 52-70

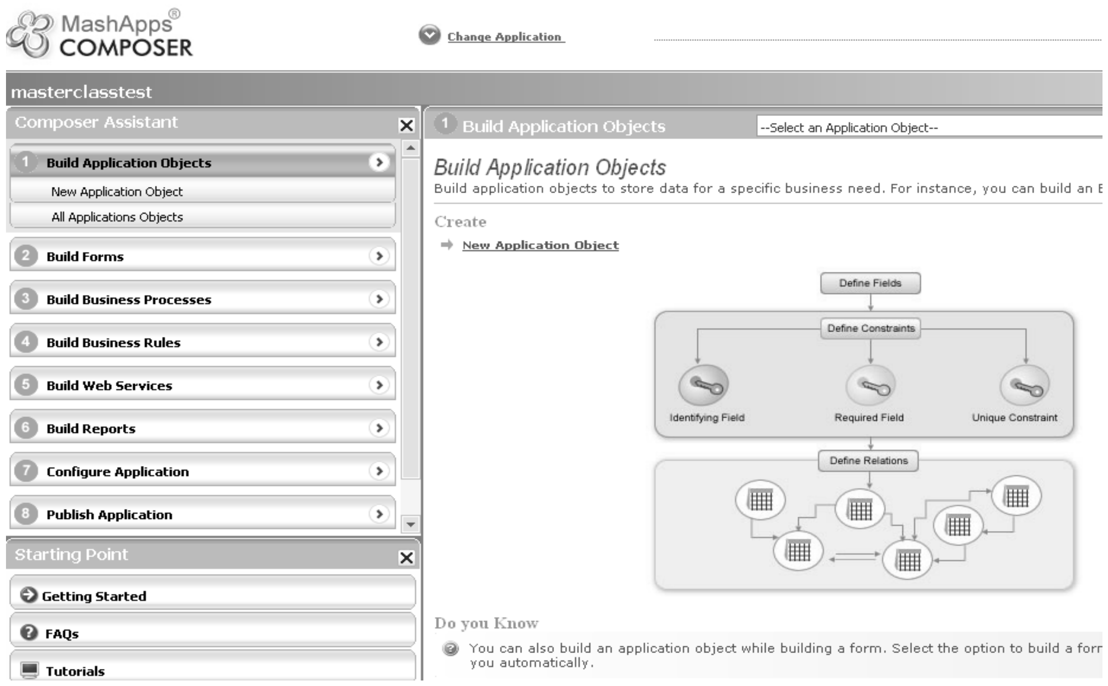
9.2 Online resources

[R1] http://www.cordysprocessfactory.com/
[R2] http://clean.ru.nl
[R3] http://wiki.clean.cs.ru.nl/ITasks
[R4] http://www.cordys.com
[R5] http://www.bpmn.org/Samples/Elements/Gateways.htm
[R6] http://www.bpmn.org/Samples/Elements/Events.htm
[R7] http://www.st.cs.ru.nl/Onderzoek/Publicaties/publicaties.html
[R8] http://clean.cs.ru.nl/download/Clean20/doc/CleanLangRep.2.1.pdf
[R9] http://www.cs.ru.nl/bachelorscripties/2009/Freek_van_den_Berg___
Creating_a_meta_model_describing_a_production_system's_concepts_and_their_relations.pdf
[R10] http://www.google.com/intl/en_us/latitude/intro.html
[R11] http://www.google.com/latitude/apps/badge/api
[R12] http://www.google.nl/search?hl=nl&rls=com.microsoft:en-us:IE-
SearchBox&q=site:http://www.google.com/latitude/apps/badge/api%3Fuser&start=270&sa=N
[R13] http://www.erikproper.eu/courses/modelleren-van-bedrijfsprocessen/practicummanual.pdf?attredirects=0&d=1
[R14] http://geojson.org/geojson-spec.html
[R15] http://www.mozilla.com/en-US/firefox/personal.html
[R16] https://addons.mozilla.org/en-US/firefox/addon/1843/
[R17] http://www.voipbuster.com
[R18] http://www.ru.nl/icis/about_icis/research_sections/model_based_system/
[R19] http://www.haskell.org/
[R20] www.json.org/
[R21] http://code.google.com/
[R22] http://code.google.com/apis/maps/index.html
[R23] http://msdn.microsoft.com/en-us/library/dd251056.aspx
[R24] http://www.omg.org/bpmn/BPMN_Supporters.htm
[R25] http://www.eclipse.org/bpmn/
[R26] http://www.webratio.com
[R27] http://sourceforge.net/projects/uenginebpmn/
[R28] http://www.softpedia.com/get/Others/Finances-Business/Cuecent-BPMN-Modeler.shtml
[R29] http://www.uml.org
[R30] http://www.petrinets.info/
[R31] http://www.json.org/example.html
[R32] http://www.cordys.com/cordyscms_com/platform_overview.php
[R33] http://www.microsoft.com/netherlands/biztalk/default.aspx
[R34] http://www.oracle.com/us/technologies/bpm/index.html
[R35] http://www.appian.com/
[R36] http://www-01.ibm.com/software/info/bpm/offerings.html
[R37] http://www.apl.jhu.edu/~hall/lisp.html
[R38] http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=clean&lang2=ghc
[R39] http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=clean&lang2=ocaml
[R40] http://strategoxt.org/
[R41] http://www.txl.ca/
[R42] http://tefkat.sourceforge.net/
[R43] http://eclipse.org/gmt/VIATRA2/

10 Appendices

10.1 *Appendix A: A CPF impression in screenshots*

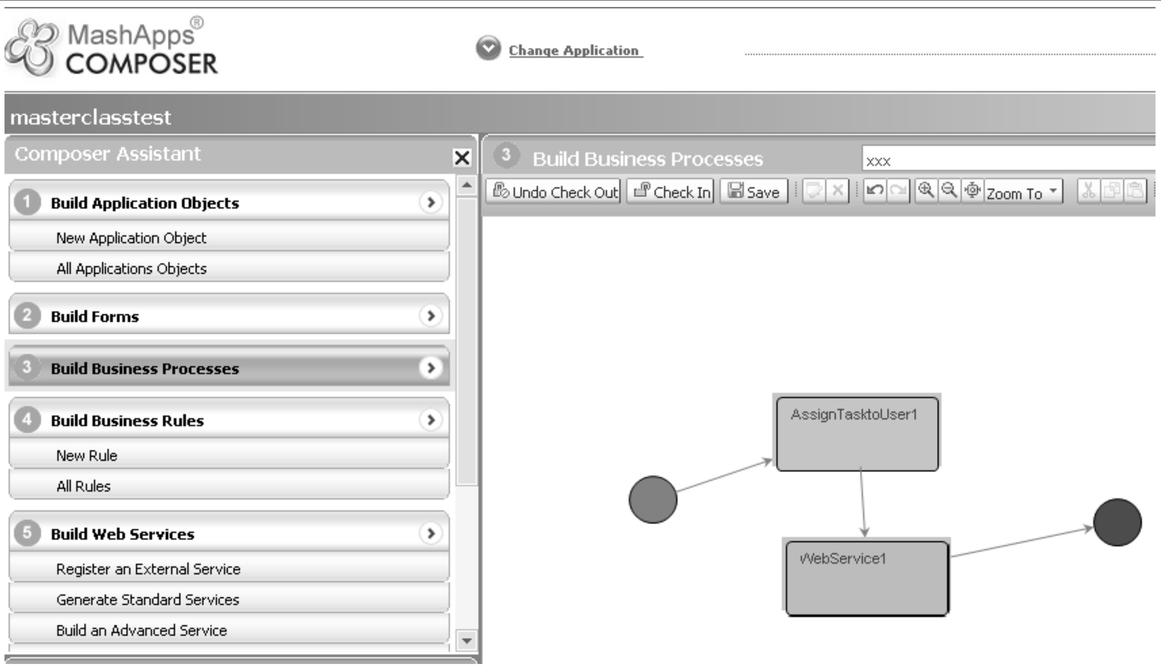
The following screenshots are meant to give an impression about what CPF is like.



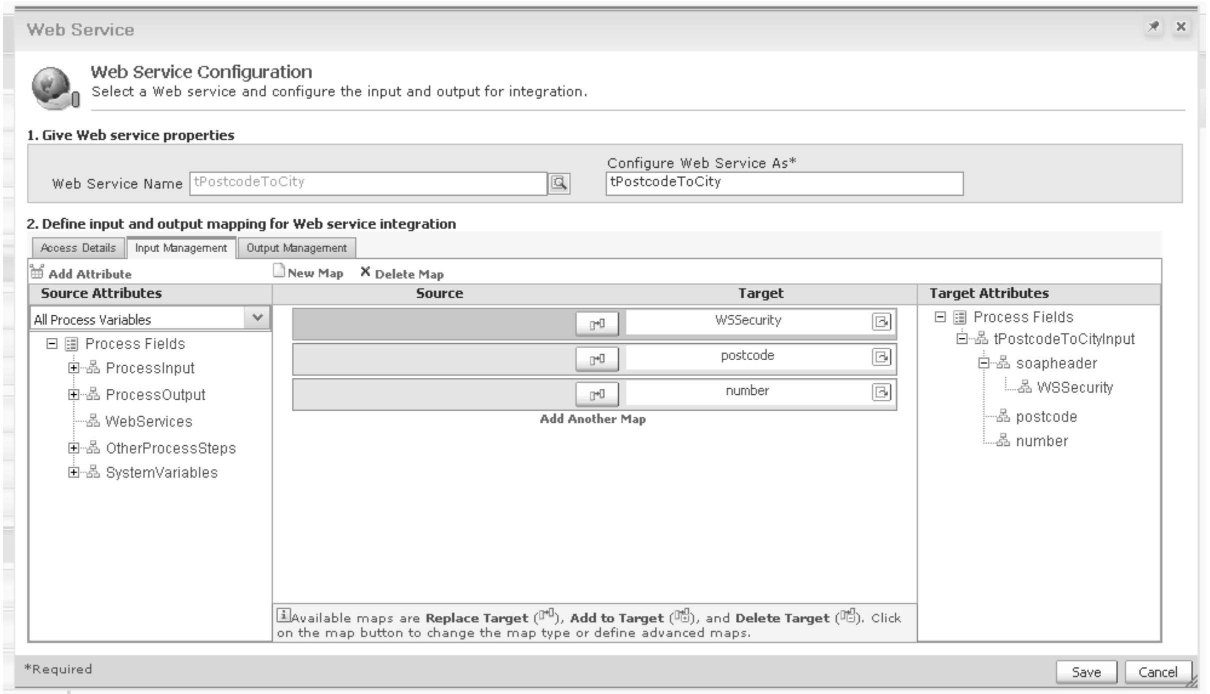
Build application objects to store data for a specific business need. For instance, you can build an E

You can also build an application object while building a form. Select the option to build a forr you automatically.

The CPF opening screen in the web browser after logging in. A whole application can be executed and designed in this interface, depending on the rights one has.



The screen in CPF in which BPMN models can be generated by mouse clicks. Double clicking on things results into a properties window, which includes the mapping of input and output. This is shown on the next picture.



Web Service

Web Service Configuration
Select a Web service and configure the input and output for integration.

1. Give Web service properties

Web Service Name: tPostcodeToCity
Configure Web Service As*: tPostcodeToCity

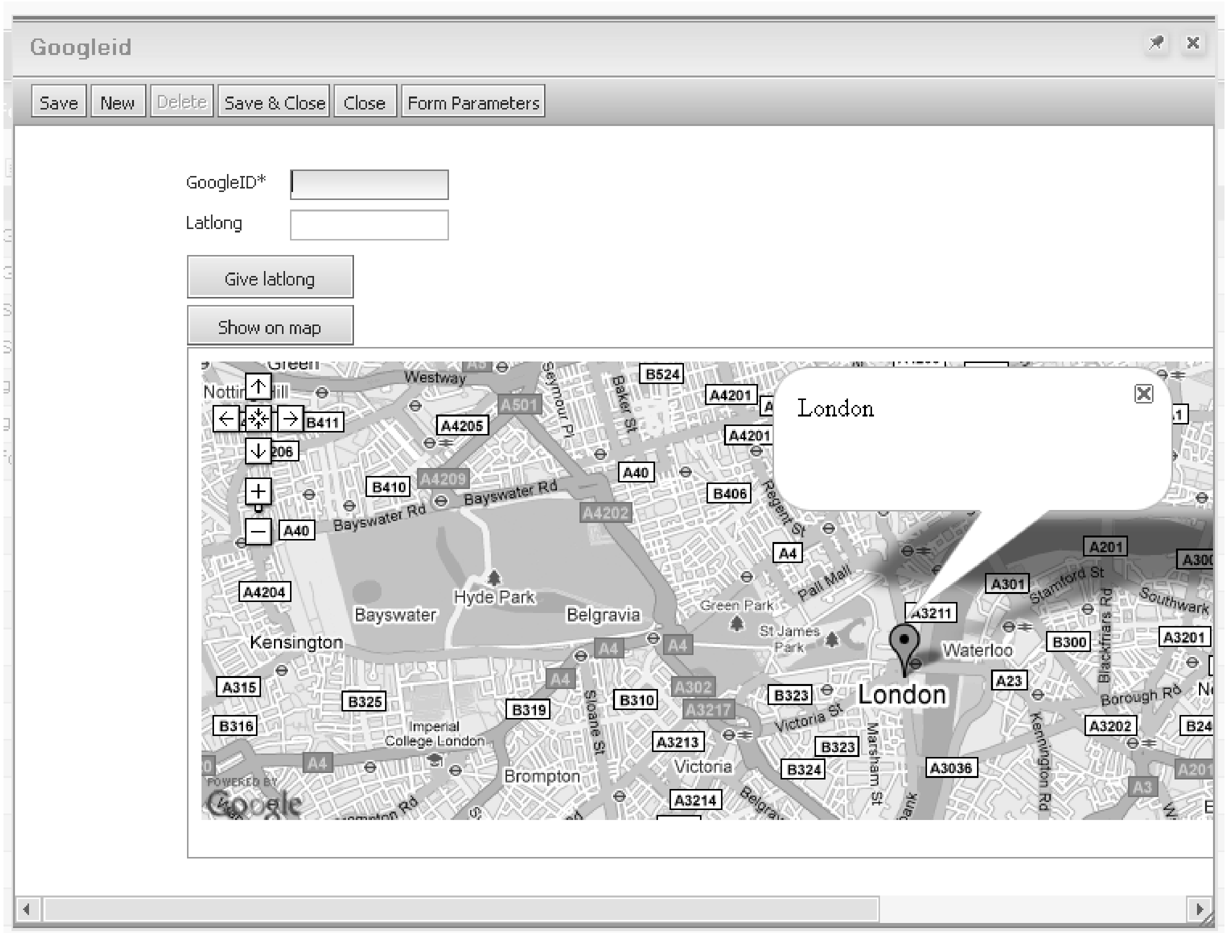
2. Define input and output mapping for Web service integration

Source Attributes	Source	Target	Target Attributes
All Process Variables		WSSecurity	Process Fields
Process Fields		postcode	tPostcodeToCityInput
ProcessInput		number	soapheader
ProcessOutput			WSSecurity
WebServices			postcode
OtherProcessSteps			number
SystemVariables			

Available maps are **Replace Target**, **Add to Target**, and **Delete Target**. Click on the map button to change the map type or define advanced maps.

*Required

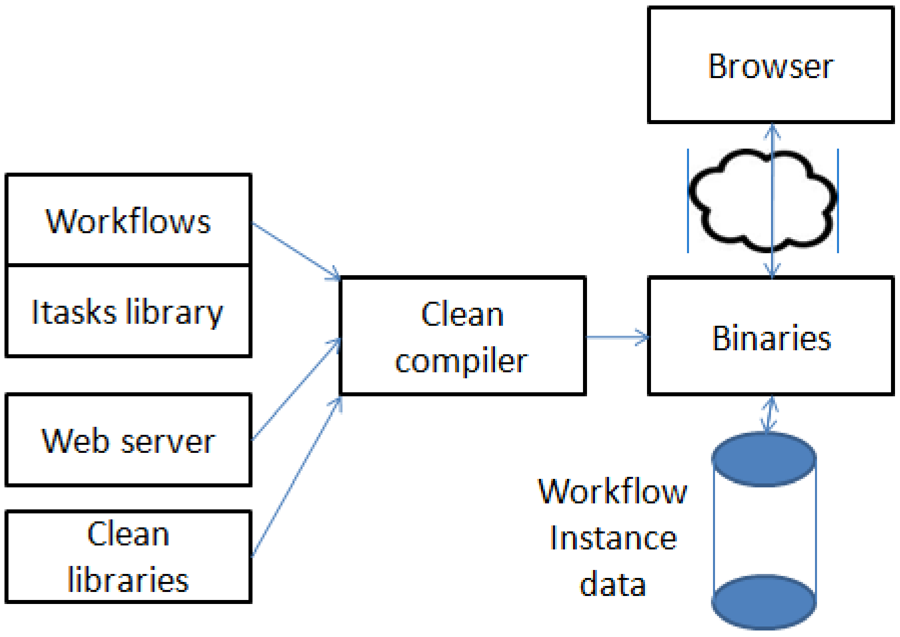
The screen in CPF in which a task (which happens to be a web service) is mapped. This means its input and output are connected to other tasks in the flow. A need for an extra layer (namely data) in addition to the standard BPMN is addressed here.



A form that has been generated in CPF. From this form business processes or single web services can be executed.

10.2 *Appendix B: A schematic overview of the Itasks architecture*

The diagrams below provides a general structure about how the infrastructure of the Itasks system looks like:



Clean is a functional programming language with its own compiler. The standard Clean libraries are compiled modularly (as needed) and in addition to this a web server and the Itasks libraries (including the desired work flows) make a run through the compiler. The leads to an application which runs on a standard PC. This application plays the Itasks server role. Through the browser the system can be approached (possibly from anywhere in the world) and as instances of the defined workflows are generated and progressed, modifications occur on the workflow instance database.

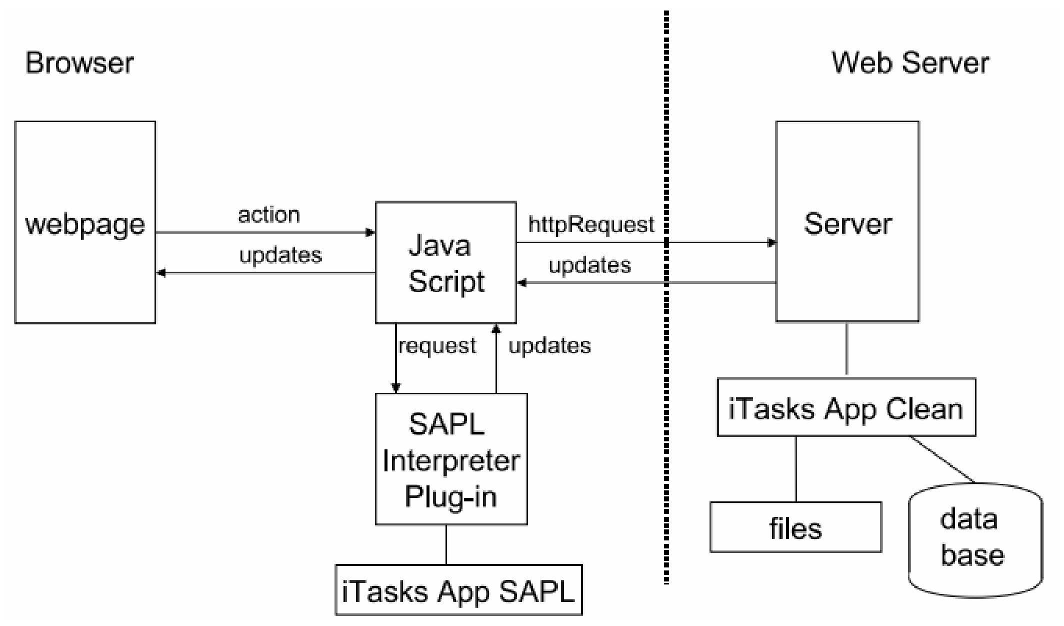
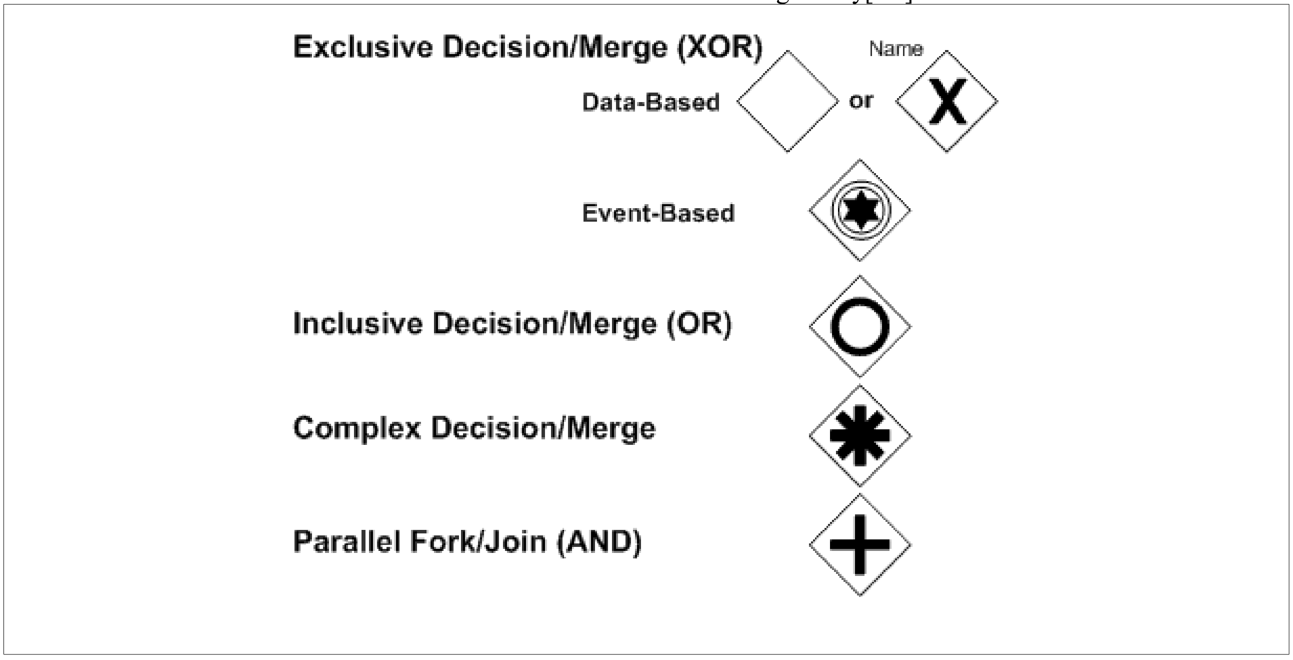


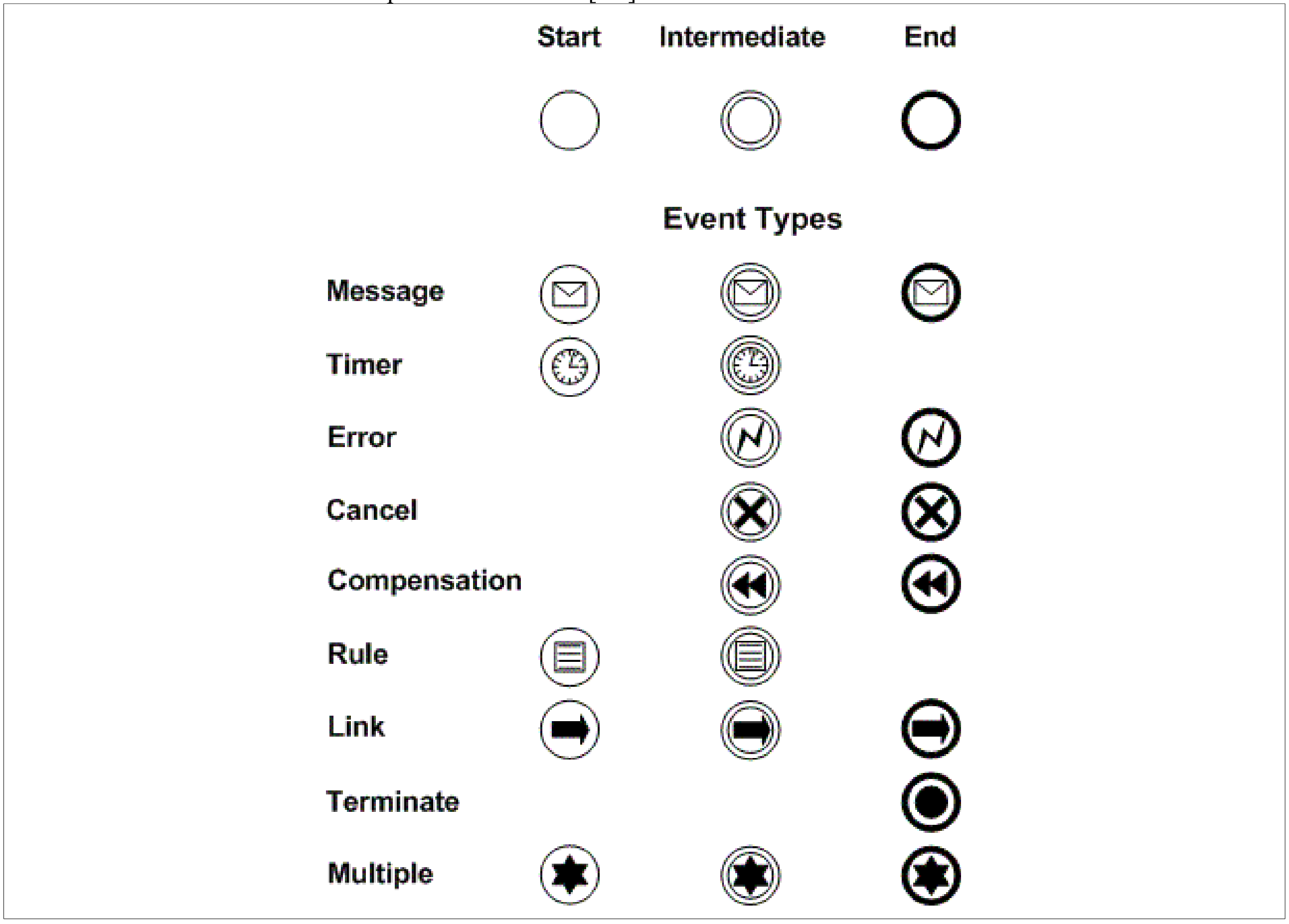
Diagram as taken from a research paper[14].

10.3 *Appendix C: Various gateways and events of the BPMN notation*

The BPMN notation offers several variations in addition to the standard gateway[R5]:



And also for events the number of options is numerous[R6]:



10.4 *Appendix D: Relevant Itasks constructs regarding BPMN*

Per Itasks construct (which is relevant to BPMN), the name, it semantics and an illustrative example are given down here, as explained carefully in two papers[11][14] and more information is at hand as well [R7]:

Name	Task a
Signature	::Task a:== *TSt -> *(a,*TSt)
Informal semantics	A task of type a, is defined as something that takes a unique task state Tst (* indicates uniqueness), and returns both a value a and a new unique state.
Example	Task42 :: Task Int Task42 = return 42;
Explanation	The just given example returns a task that always has integer value 42 as its outcome.

Name	return
Signature	return:: a -> Task a
Informal semantics	Return is a statement that lift any value in the Clean system to a task that returns that value. The advantage of doing so, is that tasks can be plugged into workflows, whereas just a value is meaningless in that context.
Example	Task42 :: Task Int Task42 = return 42;
Explanation	The just given example returns a task that always has value 42 as its outcome.

Name	updateInformation
Signature	updateInformation :: question a -> Task a | iTask a & html question
Informal semantics	The updateInformation creates a task editor, which makes it possible for a user to edit something of type a (which is given as input) as many times as possible. When the user is done editing, he can press the okay button, after which the result is returned in the form Task a. The updateInformation takes two parameters: - question: A prompt that is shown to the user - a: An input value of type a The restriciton iTask a, indicates that a has to be of some sort of kind to make the task editor deal with it and for question html means that the question should be printable. Details of this are omitted here.
Example	Task42times2 :: Task Int Task42times2 = updateInformation "Multiply this number by 2" 42
Explanation	The just given example returns a task that asks the user to multiply the number 42 by 2 by editing the field with number 42 in it.

The following combinators take multiple tasks as input and lead to a new task as a result. The combinators are used in a infix notation. By using these combinators a task starts to consist out of more tasks. This makes it possible to see a task as something that can be a workflow or business process (although in Itasks a workflow is seen as a task with some extra meta information such as its name):

Name	>>=
Signature	(>>=) infixl 1 :: (Task a) (a ->Task b) -> Task b | iTask b
Informal semantics	A task is executed and after it has been finished, the result is passed on to be used in the execution of a new task. The >>= takes two parameters: Task a: A task of type a to be executed first a -> Task b: A function that takes something of type a and is able to return this in a new task of kind b to be executed

Example	processRentalTask :: Task Void
	processRentalTask =
	  returnValidRentalTask
	  >>= \rental -> readRentalDB
	  >>= \rentalDB -> addRentalToRentalDB rental rentalDB
	  >>= \newRentalDB -> writeRentalDB newRentalDB
	  >>= \anyvariable -> return Void

Explanation	This example runs several tasks of which the results depend on each other ValidRentalTask returns a rental, which is through a lambda (written as \ in Clean) reused later on. Also readRentalDB returns something and the task addRentalToRentalDB takes the rental and rentalDB and return an updated rentalDB. Eventually the return Void is used to indicate that the whole task on its own is not supposed to return anything.

Name	>>|
Signature	(>>|) infixl 1 :: (Task a) (Task b) -> Task b | iTask b
Informal semantics	The statement taskA >>| taskB is syntactic sugar for taskA >>= \resultofA -> taskB and basically indicates that the result of taskA is not important and thrown away directly (since resultofA is not accesible in the former statement).

Example	welcomeTask :: Task Void
	welcomeTask =
	  editTask "Enter your name" ""
	  >>| editTask "Oops, I forgot your name" Void

Explanation	This example asks the user to enter its name, but 'accidently' throws away this information and thus is not able to reply with "Hello [name]".

Name	-||-
Signature	(-||-) infixr 3 :: (Task a)(Task a) -> Task a | iTask a
Informal semantics	The statement taskA -||- taskB indicates that taskA and taskB can be run interleaved in any order and once one of them finishes the result of the corresponding task is the result of the composite task. This means that taskB -||- taskA is semantically equivalent to the just mentioned statement

Example	EmployeeofthemonthTask :: Task String
	EmployeeofthemonthTask =
	  askEmployeeAforitsname -||-
	  askEmployeeBforitsname -||-
	  askEmployeeCforitsname

Explanation	This example asks three employees to enter their name and press okay. The employee who is fastes in doing so, is employee of the month.

Name	||-
Signature	(||-) infixr 3:: !(Task a) !(Task b) -> Task b | iTask a & iTask b
Informal semantics	The statement taskA ||- taskB indicates that taskA and taskB can be run interleaved in any order. However the total task is only finished when taskB has finished and this will be the result of the total task.
Example	TestingTranslationSkills :: Task String TestingTranslationSkills = ShowDutchText ||- EnterEnglishText
Explanation	A Dutch text is shown the to the user in the above example, after which the user is supposed to provide an English translation. Even though ShowDutchText as a task might finish write away (basically since no work needs to be done after it is shown), the ||- keeps it displayed on the screen.

Name	-&&-
Signature	(-&&-) infixr 3 :: (Task a)(Task b) -> Task (a,b) | iTask a & iTask b
Informal semantics	The statement taskA -&&- taskB indicates that taskA and taskB can be run interleaved in any order and once both finish their results are put in a tuple and returned as a composite task. The order of the tasks thus determines the order of the values in the resulting tuple, which means taskB -&&- taskA has different semantics and the above statement.

Example	EmployeeList :: Task String
	EmployeeList =
	  askEmployeeAforitsname -&&-
	  askEmployeeBforitsname -&&-
	  askEmployeeCforitsname

Explanation	This example asks three employees to enter their name and press okay. Once they have all done this, a next tuple construction with their names is the final result.

Name	Sequence
Formal semantics	sequence :: String [Task a] -> Task [a] | iTask a
Signature	Performs a list of tasks in sequence and returns their results in list form
Example	Calculation :: Task [Int] Calculation = sequence [return 3+3, return 5, return 8+2]

Explanation	This example performs 3 calculations in sequence and puts their results in a list. In concrete the result looks as follows (with the results of the above calculations underlined):
	`(Task (TaskDescription "" (Note "") GBFixed) Nothing`
	`(mkSequenceTask`;90 (<lambda>[line:89];46;29 (iTask; (gPrint_s;`
	`gPrint_s;190) (gParse_s; gParse_s;214) (gVi`
	`sualize_s; gVisualize_s;251) (gUpdate_s; gUpdate_s;300) (TypeCons`
	`(TypeCodeConstructor TC_Int (predefined;76 0 "TC_Int")))) [(Task`
	`(TaskDescription "return" (Note "") GBFixed) Nothing`
	`(mkInstantTask`;80 (<lambda>[line:32];37;25` **6006**`))),(Task`
	`(TaskDescription "return" (Note "") GBFixed) Nothing`
	`(mkInstantTask`;80 (<lambda>[line:32];37;25` **5048**`))),(Task`
	`(TaskDescription "return" (Note "") GBFixed) Nothing`
	`(mkInstantTask`;80 (<lambda>[line:32];37;25` **802**`)))])))`

Name	EitherTask
Signature	eitherTask :: !(Task a) !(Task b) -> Task (Either a b) | iTask a & iTask b
Informal semantics	The eitherTask returns the result of the task whichever finished first. In case task a finishes first, the result will be LEFT x (with x being the return value of task a) and in case task b finishes first RIGHT y (with y being the return value of task b)
Example	enterInt :: Task Int enterInt = enterInformation "Give me a number" enterString :: Task String enterString = enterInformation "Give me a text" testTask :: Task Void testTask = eitherTask enterInt enterString >>= \input -> showMessageAbout "" input
Explanation	This example lets the user type either a string or an integer, and displays the result of whichever one has been entered.

The following combinator deals with assigning tasks to a different user in the Itask system:

Name	@:
Signature	class (@:) infix 3 w::w (String,Task a) -> Task a -> iTask a
Informal semantics	The statement Pete@("Finish this",workTask) assigns the task workTask to worker Pete.
Example	PeteDoesWorkTask :: Task a -> Task a PeteDoesWorkTask task = Pete@("Finish this", task)
Explanation	This example delegates the work to Pete by putting the task in his inbox and adding a label "Finish this" to it.

Date and time functionality is available in Itasks as well, which are very useful for time based triggering. Some functions are:

Name	WaitForTime
Signature	WaitForTime :: !Time-> Task Void
Informal semantics	Returns a task of type void whenever the time that has been given as input parameter has been reached.
Example	WakeMeUpAtSomeTime :: Time -> Task Void WakeMeUpAtSomeTime time = WaitForTime time >>| showMessage "Wake up!!!" >>| return Void
Explanation	This example gives the user a 'wake up' notification when the input time has been reached

Name	WaitForDate
Signature	WaitForDate :: !Date-> Task Void
Informal semantics	Returns a task of type void whenever the date that has been given as input parameter has been reached.
Example	ShowMeHappyBirthday :: Date -> Task Void ShowMeHappyBirthday bday = WaitForDate bday >>| showMessage "Happy Birthday!!" >>| return Void
Explanation	This example gives the user a birthday wish when his birthday has arrived.

Name	WaitForTimer
Signature	waitForTimer :: !Time-> Task Void
Informal semantics	Returns a task of type void whenever a certain amount of time has passed.

Example	TakeA10MinuteNap :: Task Void
	TakeA10MinuteNap =
	  waitForTimer tenMins
	  >>| showMessage "Work time!!" >>| return Void
	where
	  tenMins = { hour=0, min=10, sec=0 }

Explanation	This example gives the user 10 minutes to take a nap, after which he is notified with a work time message.

The Itasks functionality just shown can be extended by combining the combinators with the Clean language, resulting in the following constructs:

allTasks:: [Task a] ->Task [a] | iTask a
Similar to the -&&- but for any number of tasks instead of two of them. The types of all the tasks need to be the same however. Can be defined recursively using the -&&- operator.

anyTask:: [Task a] ->Task a | iTask a
Similar to the -||- but for any number of tasks instead of two of them. Can be defined recursively using the -||- operator.

enterInformation::question ->Task a | html question &iTask a
Similar to the updateInformation constructor. No input value is expected besides the question though and therefore this task can be defined in terms of the updateInformation task with a default value for type a as input.

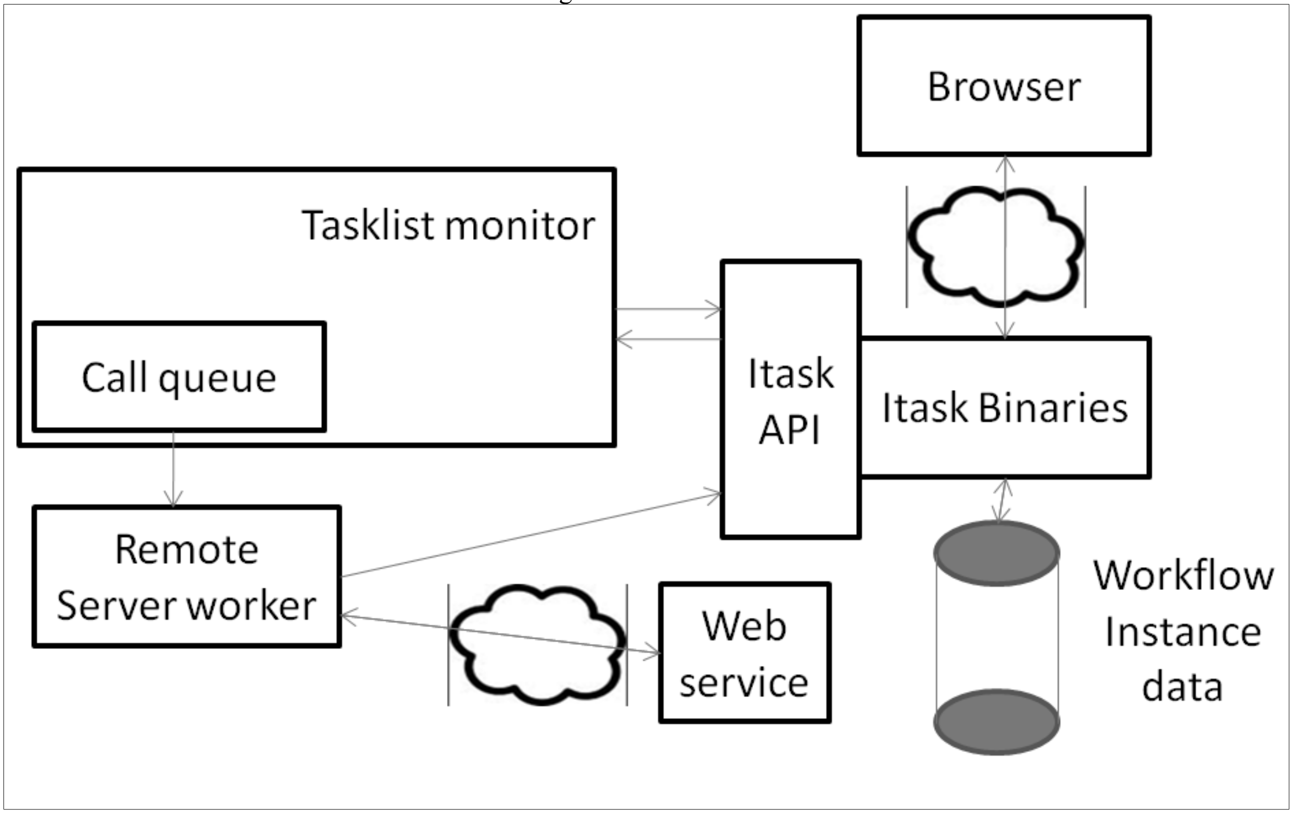
showMessage::message -> Task Void |html message
Shows a message to the user. Can be defined as an updateInformation task of type Void, since nothing needs to be edited.

showMessageAbout::message a -> Task Void |html message & iTask a
Shows a message to the user. Can be defined as an updateInformation task of type Void, since nothing needs to be edited. Additionally to showMessage, something of type a is shown.

showStickyMessageAbout::message a -> Task Void |html message & iTask a
Shows a message to the user. The main difference with showMessageAbout is that the okay button is not shown and that it can easily be binded with a follow up task which is displayed in the same screen.

10.5 *Appendix E: Executing a web service in the Itasks system*

In order to be able to use RPC in Itasks the following intrastructure is introduced:



The components 'Tasklist monitor', 'Call queue' and 'Remote Server worker' are added and have been written in JAVA, due to the inability of Clean to support multithreading. In order to have a RPC executed and its result succesfully returned the following steps take place:

- Somewhere in the Itasks code a RPC call is made during runtime (thus in the Binaries) in a task. This task is put on hold, until the whole result has been returned
- The Tasklist monitor polls at a certain interval (eg 2 seconds) whether Itasks still has RPCs that need to be handled, by asked the Itask API
- If so, the tasks (including the one that is discussed here) are returned by the Itask API
- The Tasklist monitor adds the task to the queue
- The Remote server worker picks up tasks from the queue and access the needed web service.
- The web service returns a result after which the Remote server worker hands it over to Itasks through the API in base64 format (in order to make it transferable).
- Itasks delivers the result at the right place, after which the task that has been put on hold can continue its activities

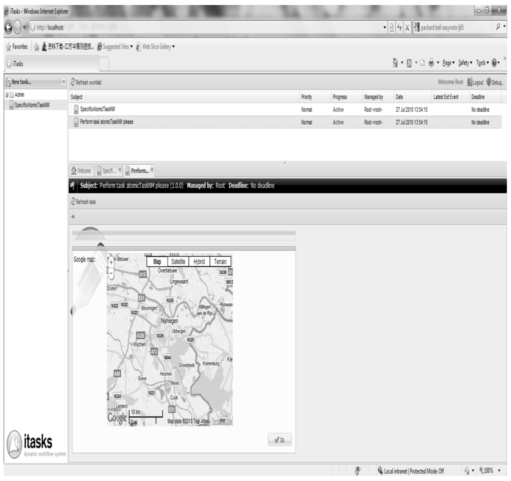
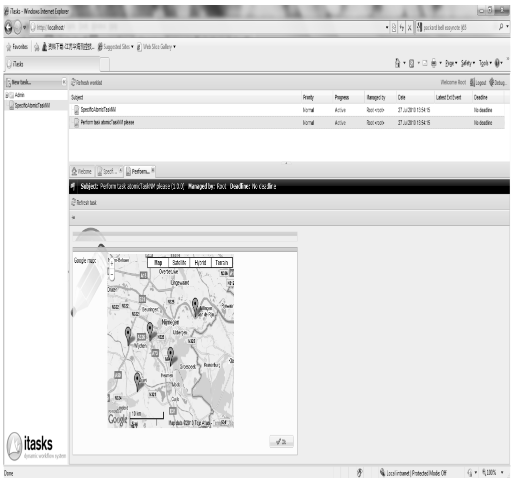
The JAVA extension called Deamon is part of the standard libraries of Itasks and is thus included in the start distribution and written by Erik Crombag[R3].

10.6 Appendix F: A demonstration of a Google maps task in Itasks

This appendix will show that an atomic activity can be more then just transforming input to output. For the next activity the input is considered Void, while the output is of kind GoogleMap. This leads to the following code (by exactly copying the implementation of an atomic task and only changing the input and output type):

```
module googleMaps

import iTasks, GeoDomain

:: AtomicTaskNMOutput = { googleMap :: GoogleMap }
derive class iTask AtomicTaskNMOutput
derive bimap (,), Maybe

AtomicTaskNM :: UserId Void -> (Task AtomicTaskNMOutput)
AtomicTaskNM user input =
 user @:("Perform task atomicTaskNM please",
          showStickyMessageAbout "" input ||- (enterInformation ""))

Start :: *World -> *World
Start world = startEngine workflowitems world
where
       workflowitems = [ workflow "SpecificAtomicTaskNM"
                             (AtomicTaskNM "root" Void)
```

When the above code is executed and the task initiated the following screenshots illustrate what happens:

A Google Map is shown at which one can put as many markers as one wishes, just by clicking on the map. Zooming in and out is possible, as well as all the functionality Google Maps normally offers. The above example illustrates that the process of creating output values can be much more then a one way process from the user to the system. Instead of this, the generation of data can be any interactive process one can imagine, from a Google Maps selection till a whole working day from someone at the office. There is a paper[57] describing interactive extensions for Itasks.

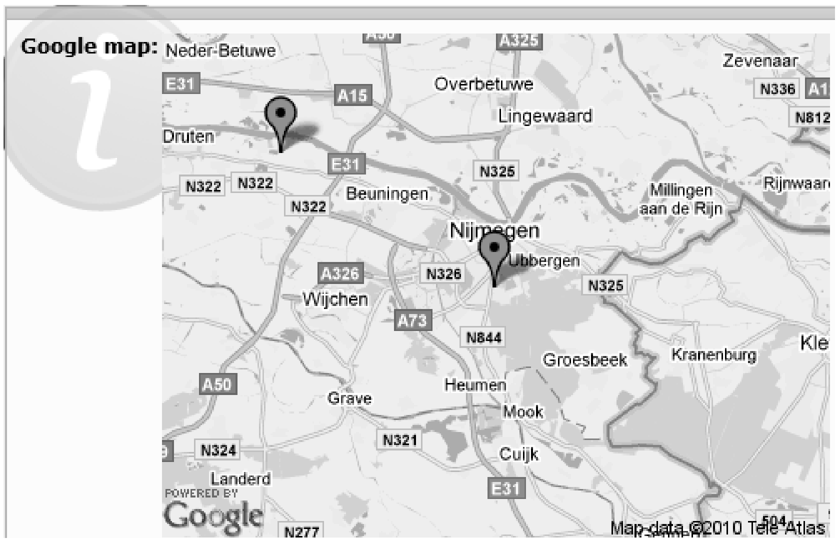
After clicking on the map twice and pressing okay, the resulting data type might look as follows:

```
{ AtomicTaskNMOutput | googleMap = { GoogleMap | center = (51.82, 5.86), width = 400,
height = 300, mapTypeControl = True, navigationControl = True, scaleControl = True,
scrollwheel = True, draggable = True, zoom = 10, mapType = ROADMAP, markers =
[{ GoogleMapMarker | position = (51.8870114574153, 5.6828454589844), infoWindow =
{ GoogleMapInfoWindow | content = "", width = 0 } }, { GoogleMapMarker | position =
(51.819151114077, 5.8586267089844), infoWindow = { GoogleMapInfoWindow | content = "",
width = 0 } }] } }
```

Using the following code and inserting the above definition, one can then show the map again:

```
module googleMaps

import iTasks, GeoDomain

:: AtomicTaskNMInput = { googleMap :: GoogleMap }
derive class iTask AtomicTaskNMInput
derive bimap (,), Maybe

AtomicTaskNM :: UserId AtomicTaskNMInput -> (Task Void)
AtomicTaskNM user input =
 user @:("Perform task atomicTaskNM please",
         showStickyMessage "" input)

InputMap :: GoogleMap
// The definition from above goes here

Start :: *World -> *World
Start world = startEngine workflowitems world
where
       workflowitems = [ workflow "SpecificAtomicTaskNM"
                             (AtomicTaskNM "root" { googleMap = InputMap } )
                       ]
```

After executing the code and initializing the task, the following result appears:



10.7 Appendix G: A sample of the Itask API

At the current time of writing the Itasks API has not been clearly defined and documented yet. By running Itasks in the Firefox Browser[R15] and running the Firebug[R16] Add-on, one can determine which calls are made in order to provide certain functionality. This leads to the following relevant functionality with respect to this thesis:

Name	Authentice to the Itasks server and obtain a session token
URL	http://localhost/handlers/authenticate
Parameters	password: Password of the user to be authenticated username: Name of the user to be authenticated
Example	http://localhost/handlers/authenticate?password=root&username=root
Response	`{"success": true, "displayName": "Root", "sessionId": "yyytmueluufenmdphqchozycvdzkmcmv"}`

Name	Start a new workflow
URL	http://localhost/handlers/new/start
Parameters	_session: The session token in order to make sure that the one calling this handlers has the appropriate rights to perform this action workflow: The name of the workflow
Example	http://localhost/handlers/new/start?_session=nicdwmelszdbzjmiripctlfiymxhhjng&workflow=SpecificAtomicTaskNM
Response	`{"success" : true, "taskid": "6"}`

Name	Obtain the list of task instances currently running
URL	http://localhost/handlers/work/list
Parameters	_session: The session token in order to make sure that the one calling this handlers has the appropriate rights to perform this action
Example	http://localhost/handlers/work/list?_session=yyytmueluufenmdphqchozycvdzkmcmv
Response	`{"success" : true, "total" : 3, "worklist" : [{"taskid" : "2", "manager" : "Root ", "subject" : "SpecificAtomicTaskNM", "priority" : "NormalPriority", "progress" : "TPActive", "timestamp" : 1280319391, "latestExtEvent" : 1280319403, "deadline" : null, "tree_path" : [], "tree_last" : false, "tree_icon" : "task", "tree_new" : false},{"taskid" : "4", "manager" : "Root ", "subject" : "Delete users", "priority" : "NormalPriority", "progress" : "TPActive", "timestamp" : 1280321211, "latestExtEvent" : null, "deadline" : null, "tree_path" : [], "tree_last" : false, "tree_icon" : "task", "tree_new" : false},{"taskid" : "6", "manager" : "Root ", "subject" : "SpecificAtomicTaskNM", "priority" : "NormalPriority", "progress" : "TPActive", "timestamp" : 1280321254, "latestExtEvent" : 1280321267, "deadline" : null, "tree_path" : [], "tree_last" : true, "tree_icon" : "task", "tree_new" : false}]}`

Note that localhost needs to be replaced with the URL of the Itasks system. In a testing environment this system is normally ran on the local computer, therefore localhost has been shown.