wordt
NIET
uitgeleend

Variability and Web Services

Michiel Koning

21st June 2006

Rijksuniversiteit Groningen
Bibliotheek FWN
Nijenborgh 9
9747 AG Groningen

Abstract

In today's world, communication between computer systems, business-to-business communication, is becoming ever more important; just think of the travel agency where you would like to book a flight, hotel and car rental at the same time. One way to facilitate this interbusiness communication is by use of Web services, systems that make applications available via a network, such as the Internet.

The business environment is also a dynamic one. What customers want today, may no longer be easy to achieve tomorrow, or it may no longer be what they expect. Therefore flexibility is also a desirable trait of computer systems today. This flexibility can be provided or addressed by incorporating variability into a system.

This thesis discusses how variability can be incorporated into (Web-)service-based systems. It introduces a language, VxBPEL, an adaption of an existing language, able to capture variability in these systems and discusses a proof-of-concept implementation of an application platform able to interpret this language and handle the variability it contains.

Contents

1. Background

In today's world, information is exchanged between companies' computer systems on a daily basis. Consider the travel agency where you would like to book a flight, hotel and car rental at the same time; the online store where you can see the current stock for the item you want to buy; the supermarket that automatically places an order at the distributor when stocks run low. The problem with information exchange, however, is that all the computer systems are different, and that there is not one way to access all these systems uniformly.

This problem is addressed by Web services. Web services offer a uniform way to access information between systems, even when these systems run on different platforms (such as Unix, Linux, Windows and MacOS), by using standardized Web technologies. Communication takes place via the HTTP protocol, the same protocol which is used to access web sites via a web browser, because HTTP is so simple and so widely used. This can greatly simplify the communication with other computer systems, as a system need only be able to access the Web service provided by other systems in order to exchange or request information with them, instead of having to support each system's way of communication separately.

With the introduction of Web services, information exchange systems that are entirely or largely based on Web services emerged. Such systems are known as Service-Oriented Systems or Service-Centric Systems.

Also, in order for businesses to remain competitive, it is advantageous to be able to accommodate changing needs of customers quickly. One way of achieving this is by designing a system to have parts that are variable, which means they are flexible or can be changed. A system supporting these variable parts is said to have variability.

Both having support for easy information exchange and for variability are advantageous to a business. Therefore, this thesis addresses both these concepts and combines them to allow for a system that has support for both. The solution described is VxBPEL, an extension of BPEL, a widely used language to describe service-centric systems, that allows one to design a system to have variability. To show that VxBPEL indeed works, an adaptation of a platform for BPEL that supports the variability described by VxBPEL is also discussed and tested.

1.1. Running example

This section describes an example that will be used throughout this thesis. As it will form the basis for many examples and test cases, it will be discussed in-depth here. This example is taken directly from the WS-BPEL 2.0 specification ([11], section 14.3).

This example considers a simple loan approval Web Service that provides a port where customers can send their requests for loans. Customers of the service send their loan requests, including personal information and the amount being requested. Using this information, the loan service runs a simple process that results in either a "loan approved" message or a "loan rejected" message. The approval decision can be reached in two different ways, depending on the amount requested and the risk associated with the requester. For low amounts (less than $10,000) and low-risk individuals, approval is automatic. For high amounts or medium and high-risk individuals, each credit request needs to be studied in greater detail. Thus, to process each request, the loan service uses the functionality provided by two other services. In the streamlined processing available

for low amount loans, a "risk assessment" service is used to obtain a quick evaluation of the risk associated with the requesting individual. A full-fledged "loan approval" service (possibly requiring direct involvement of a loan expert) is used to obtain in-depth assessments of requests when the streamlined approval process does not apply.

A UML-like flow diagram depicting the original process is shown in figure 1.1. In this diagram, and all the other similar diagrams used throughout this thesis, a rounded rectangle represents a part of the process, called activity. The type of activity is surrounded by << >> (like `<<receive>>`) and the name is given directly below its type. The bottom half of the rounded rectangle contains extra information for this activity where needed (e.g., the `receive1` activity is annotated with `approve(request)`, meaning the `approve` operation of the process is invoked). The straight-angled rectangles containing activities denote that an activity occurs in collaboration with another party (partner), indicated by the name in the top-left corner. If an activity contains other activities, this is denoted by a rounded rectangle directly attached to an activity. Diamonds depict choices or points where two control flow paths join.

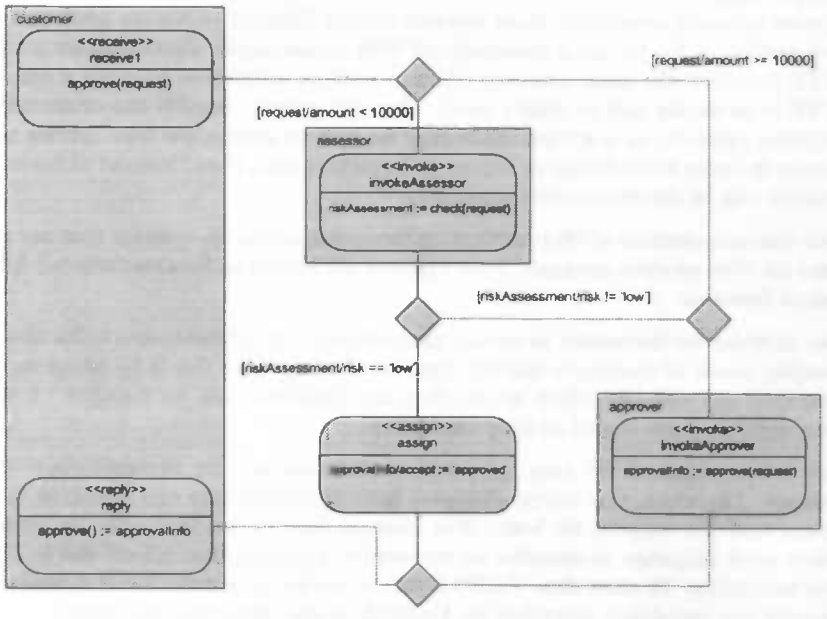

Figure 1.1.: The original example process

1.2. Structure of this thesis

The structure of this thesis is as follows. Chapter 2 gives a more thorough introduction to the concepts of variability and Web services, providing the reader a basis to understand the remainder of this thesis. Chapter 3 explains what variability modeling can mean for Web service based systems and in what ways variability can be used. It will also present the central question this thesis answers. Chapter 4 discusses a solution to combining variability and Web services: VxBPEL. In chapter 5, an implementation of VxBPEL into an existing server application (ActiveBPEL) is presented, as well as an extension to the variability modeling framework COVAMOF, allowing for a graphical representation of variability modeled by VxBPEL. In chapter 6, several test cases are presented that validate the solution presented in chapter 4. The final chapter, chapter 7 contains conclusions based on the results of these test cases.

The appendix contains a short glossary of terms, acronyms and abbreviations (appendix A) used in this thesis that are not explained when they are used. This can be referred to in case a term, acronym or abbreviation is unclear.

2. Introduction

This chapter introduces the subjects of variability and Web services. In order to do so, it discusses the concepts of variability, Web services and Web service composition, and terminology needed to understand the remainder of this thesis.

2.1. Variability

Traditionally, software has been written one project at a time. For each customer and each system, there would be a requirements, design, and implementation phase. Sometimes a product was started from scratch, other times source code originally written for a different product would be reused. This type of reuse was however never explicitly designed. It is now becoming increasingly common to design a system to consist of smaller parts, called components. Each of these components contains specific functionality. These components also supply a predefined way to access this functionality called an interface. By dividing systems into components and using these components for multiple systems, reuse has essentially become explicit[2, 9]. Using these interfaces and the fact that interfaces do not change, different components can be combined to interact with each other and together form one system. This is called a composition of components.

However, most systems will need to be adapted in their life cycle. This can be due to customers having new wishes for the system (i.e., changing requirements), because of compatibility, because of new developments, etc. In short, it is desirable changes can be made to the systems. Building your system out of components also simplifies this. Because the functionality of a component is defined by its interface, it is possible to freely change the internals of such a component, altering its behaviour, as long as the functionality it provides conforms to the interface specification. It is also possible to change the outward behaviour, by changing the interface (and thus the functionality provided). Also, by combining different components, one can create a different system altogether. When the information about the ability to change a system is explicit, it is called "variability"[1].

The next subsections will discuss several concepts used in variability research. A number of these concepts are specific to, or described as defined in COVAMOF[16]. COVAMOF (ConIPF Variability Modeling Framework) is a variability modeling approach that is able to model variability on several levels of abstraction and has explicit modeling for many of the concepts described in the next subsections. It models variability generically, allows to manage complexity of variability and enables automation in the variability process.

2.1.1. Internal and external variability

There are two types of variability, which we will refer to as "internal" and "external" variability. Internal variability is variability within a component that doesn't change its conformance to its interface (unless, of course, the interface is changed as well). This could encompass employing a different algorithm internally. External variability is the variability within a composition. Examples would be swapping a component for a different one, or adding more components.

As an illustration, consider the following example. A system needs to access information from a data source, say, an ordering system that requires access to ordering history. A component could offer functionality such as searching for an order by several fields (like date and customer).

The system does not need to know how this information is stored. It could be represented by a database, or it could also be represented by a set of files in the file system. The functionality (searching in and accessing orders) stays the same, but the behaviour (accessing files or a database) differs.

If there are two components implementing this interface, one accessing files, one accessing a database, we can build two systems by choosing one of these two components for each system. They offer the same functionality, the system has the same composition, but its behaviour is still different. This is internal variability.

An example of external variability: suppose a system is built of a set of components which supply interfaces and require others. If we now replace one of these components with a different one that has more features, but also has more required interfaces (requiring additional components to be added), we can vary the features this system supports without altering the components themselves.

2.1.2. Variation points and variants

A part of a system that can be varied is called a "variation point". The options for a variation point are called "variants" (or "alternatives"). When a variant is chosen for each variation point, the collection of these choices is referred to as a "configuration"[4].

Suppose in the example system above the component which accesses data from a set of files on the file system is chosen. There might be other components that try to access data from either a file system or a database. It would make sense for all these components to use the same type of data source, even though they can be completely unrelated otherwise, to maintain consistency in the system. Also, it is possible that two choices together are incompatible, because they conflict by themselves, or because they both influence the same property of a system in different ways (such as performance). Such a relationship between components (or variation points) is called a "dependency relation"[16] and is described in the next section.

2.1.3. Dependencies

In order to be able to model how variation points influence dependencies, it is important that knowledge about these dependencies is available. Without any knowledge, it is impossible to decide which variants to use for which variation points and define constraints for these variants.

The COVAMOF framework includes the modeling of knowledge about dependencies and variation points. It is modeled as follows[17].

In the COVAMOF model, dependencies are relations between variation points and specify a property which these variation points influence. This property could tell whether, in a certain set of variants, each variant in this set is compatible with each other variant in the set, or specify the value of a system property (also known as quality attribute) such as performance. It might not always be directly known how a dependency influences this property. This is related to the type and availability of knowledge about this property.

There are different types of knowledge. The first type is *formalized* knowledge. This is explicit, well-known knowledge that is written down in a formal language and can be used and interpreted by computer applications. A different type of knowledge is *tacit* knowledge. This is knowledge that exists in the minds of engineers involved with the system, but is not written down. The final type of knowledge is *documented* knowledge. This is knowledge that is expressed in informal models and descriptions.

Suppose there is a system which has a variation point with variants A and B, and a dependency "required memory". In this system, an example of formalized knowledge would be: "selecting

variant A will always add 24 MB to required memory and selecting variant B will always add 12 MB to required memory" (of course written down in a formal language). In this case it is well known how a certain variant (i.e., A or B) influences a specified dependency (i.e., required memory). Tacit knowledge is implicit knowledge that is used to, for example, steer a complicated performance issue with many trade-offs into a certain direction. Only experts, which have solved many of these trade-off issues, can make an educated guess of how to do this; someone who has never seen the issue before will most likely need a trial-and-error approach to find the configuration that best suits the performance expected. Documented knowledge can be exemplified by documentation that only specifies performance and memory usage for several common, complete configurations. This gives no explicit information about other configurations.

When knowledge is modeled as in the COVAMOF model, choices can be made supported by this knowledge, or in the case of formalized knowledge, even automatically. An advantage of making as much information as possible explicit by documenting or formalizing it is that one is no longer as dependent on system experts as when there is little or no knowledge made explicit.

2.1.4. Realization relations

Systems can contain variation points at several levels of abstraction. An example would is a reservation system. In this system a service is invoked to make a reservation. It can be specified that this service is a variation point and has two variants. This is a high level view. However, when one looks at the actual implementation of the system, which is a lower level view, it could be that several variation points are introduced which all have several variants to allow for the option at the higher level. It could be that the two variants are incompatible with each other, because the two services require different messages, and therefore require extensive changes in the implementation to switch from one variant to another.

Figure 2.1 depicts this example. It uses COVAMOF's notation for variation points and realizations: a variation point is denoted by a circle, its associated variants as triangles attached to this circle, and the dotted line separates the VPs on the lower level from the VP on the higher level.

In choosing the variant for the reservation service on a higher level, the variation points on a lower level (i.e., all the parts that interact with the message that is sent to the service) need to have a certain configuration to realize the higher-level variation point. Such a relation between variation points is called a "realization relation"[16]. A realization relation, when formalized, can however improve the manageability of the variability a system has, because the exact details of which variation points allow which other variation points to exist can then be determined automatically. In other words, one need only be concerned with the variation points at the highest level of abstraction when realization relations are known and formalized.

2.1.5. Binding time

There are several stages in the life cycle of a software system where a configuration may be selected, after which it is no longer possible to change the configuration selection. This selection is called "binding", and the stage in the life cycle at which binding occurs is called "binding time". Binding can occur at several stages, e.g. the compilation stage, the installation/deployment stage or at run-time. When binding occurs in a certain phase of the life cycle, the system configuration is fixed as of that phase. Run-time binding is different. When a choice is made at run-time, and binding occurs, it is possible to allow this binding to be redone; in other words, it allows rebinding at run-time. Being able to rebind variation points at run-time means that a system can reconfigure at run-time without shutting down. This is especially interesting for systems in a dynamic environment such as those based on Web services, as this means they can quickly adapt to respond to changes.

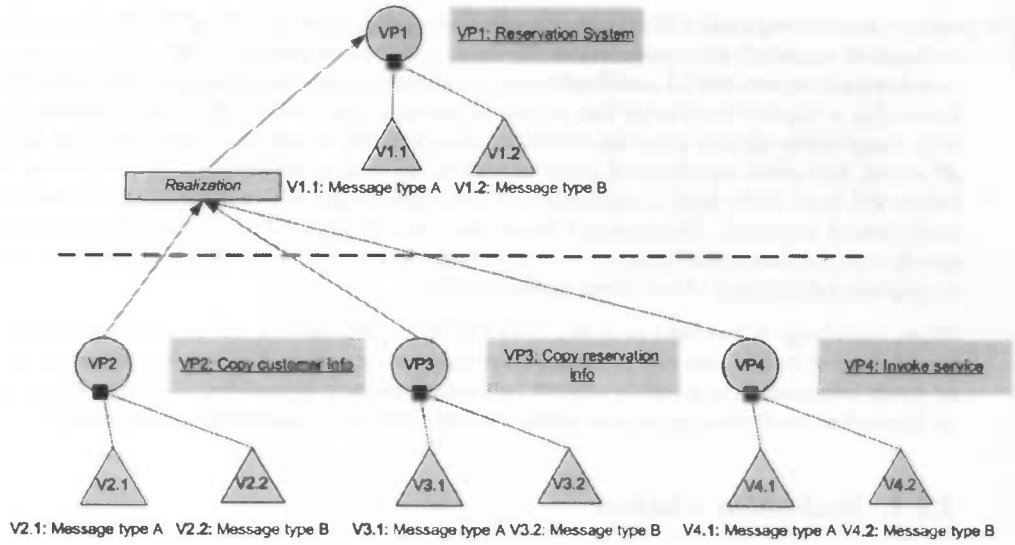

Figure 2.1.: The choice at the architecture level prescribes choices on the implementation level

2.2. Web services

Over the last years, many information systems have started to use the Web to exchange information across the Internet or corporate intranets. These information systems are now developed (or adapted) to have a Web-interface or be a Web-based system to accommodate this trend.

A Web service, according to the W3C, is "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."[23] In other words, it is a software system that allows machines to collaborate over a network, be that the Internet or an intranet, using standard formats and protocols (e.g., SOAP, HTTP, XML). The use of these standards ensures that two machines that understand the standards can interact, no matter how they are implemented (hence the term "interoperable"). The interface of such a software system is described by WSDL, another Web standard.

A Web service is only a characterisation of a resource that specifies functionality that is provided; as such they are actually very similar to interface definitions. There is also the notion of an "agent": a concrete implementation of this abstract definition.

Web services can be seen as special software components that are located, bound and executed at run-time using the aforementioned standard internet protocols (UDDI, WSDL and SOAP)[6]. By combining several services, one can form a loosely-coupled architecture that allows for great flexibility at run-time. Because services have explicit interfaces, it is relatively easy to integrate third-party services into such an architecture. Also, explicit interfaces allow services to be substituted at runtime[19, 15].

An overview of the Web service architecture is shown in figure 2.2. Basically, services register with the UDDI repository and list information about themselves. When a company's system wishes to use a service, it contacts the repository with a service search via a network (e.g., the Internet). The repository returns a set of results, the system chooses one of the services to use, and contacts it through the same network.

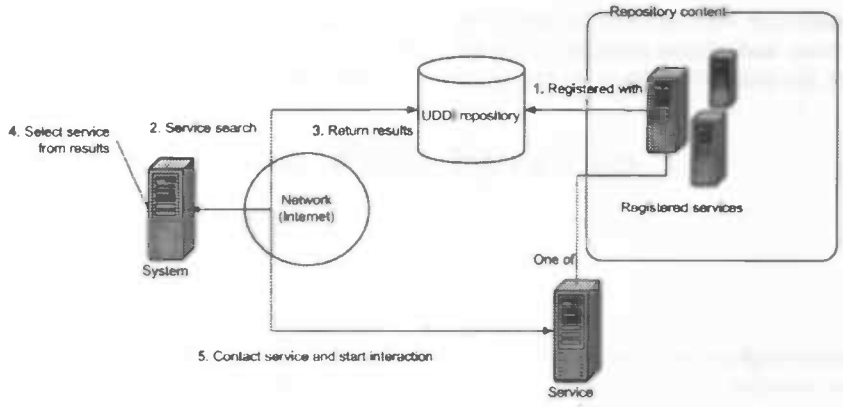

Figure 2.2.: The Web service architecture

2.2.1. Web service orchestration and Service-Oriented Architectures

Just like software components are combined to form one system, a composition of Web services in a process flow can be used to define what is called a Web-based system. Two terms are used in relation to this: orchestration and choreography[14]. Orchestration describes how Web services can interact with each other at the message level, including the business logic and execution order of the interactions. These interactions may span applications and/or organizations, and result in a long-lived, transactional, multi-step process model. Choreography tracks the sequence of messages that may involve multiple parties and multiple sources, including customers, suppliers, and partners. Choreography is typically associated with the public message exchanges that occur between multiple Web services, rather than a specific business process that is executed by a single party.

Although the line is thinning due to recent enhancements and standards convergence, there is an important distinction between orchestration and choreography. Orchestration refers to an executable business process, possibly with interactions with both internal and external Web services. This process is always controlled from the perspective of one of the business parties. Choreography is collaborative in nature, in which each party involved in a process describes the part they play in the interaction. In other words, there is no clear control flow defined in a choreography, but a party merely defines what message exchanges with that party will occur in the interaction.

A service-oriented architecture (SOA) is an architecture that uses services in general (not specifically Web services) to support the requirements of the users of the software. All the service components in a SOA are highly reusable, and as such allow faster adaptation to changing requirements.

2.2.2. An orchestration language: WS-BPEL

An example of a language standard for orchestration is WS-BPEL 2.0 (Web Services Business Process Execution Language)[1]. It is a continuation of the business process language BPEL4WS (Business Process Execution Language for Web Services) originally introduced by IBM, Microsoft and BEA. It provides a notation and semantics for specifying business process behaviour based on Web services. A process is defined in terms of its interactions with partners. A partner may provide services to a process, require services from a process, or interact two-way with a process. BPEL

[1]This standard is commonly referred to as BPEL and as such, WS-BPEL, BPEL4WS and BPEL will be used interchangeably throughout this thesis.

orchestrates Web services by specifying the order in which it is meaningful to call a collection of services, and assigns responsibilities for each of the services to partners. It can be used to specify both the public interfaces for the partners and the description of the executable process.

3. Variability and Web Services

3.1. Why variability?

Most non-service based software systems are static. As such, run-time configuration of these systems is limited or not possible at all. This means that if such a system supports variability, it is configured once, in a certain phase of its life cycle, after which it is fixed. If a system is to change at run-time, this means it will have to be shut down to be reconfigured. This is different for dynamic systems such as (Web) service based systems.

Service based systems are loosely coupled because service definitions in WSDL are merely abstract definitions. These service definitions can therefore be compared to interfaces of components. Binding of these definitions to concrete services can be done at run-time. This means that service based systems of themselves already offer extreme flexibility. However, there is as of yet no notion of explicit variability modeling in system definitions, even though, given the dynamic environment in which service based systems usually operate, being variable and having variability explicitly modeled presents a great advantage. For example, most Web service providers are bound by a SLA (Service Level Agreement, a sort of "contract" between two parties) to provide a certain level of Quality of Service (QoS). Deviating from this QoS requirement could result in penalties or loss of customers. The fluctuation and/or loss of QoS is very much an issue in the dynamic environment in which Web services are found. Examples of matters that could result in a loss of QoS are network irregularities (services may become unreachable due to fluctuations in available bandwidth and throughput rates) and third party services that may be unreliable and as such can fail unexpectedly[15].

Modeling variability explicitly means that loss of QoS due to a failing third party service could be countered by having several back-up services explicitly defined in the process definition as a variation point, and altering the configuration of this variation point (possibly automatically) when it turns out the currently configured service fails. This is an advantage because it means it is possible to conform to an SLA where normally you would have failed to comply to your obligations in the SLA.

Another advantage is that it is possible to vary QoS for each request. A customer could specify what level of QoS would be required for the request, and before this request is processed, the configuration of the system is decided for this one request. The advantage here is that there is a single definition of this system possibly allowing numerous levels of QoS by merely altering a few variability parameters of the system.

3.2. Requirements for variability

In order to allow the flexibility and loose-coupledness service based systems offer, to address, for example, QoS concerns, it should be possible to support several types of variability. These are[18]:

- Replacing a service by a different one with the same interface. If a service performs inadequately or is unreachable for whatever reason, another service can be transparently used.

- Replacing a service by one with a different interface. This is essentially the same as the previous type, but with the note that the interface can be different and therefore will not be entirely transparent. This can include changing the protocol used to access the service (e.g., REST instead of RPC).
- Changing the parameters with which a service is invoked. If a service provides a QoS management interface, one could change the QoS parameters of the invoked service dependent on the QoS contract with the customer.
- Changing the composition of the system. If normally several services are invoked in parallel to cross-verify results, and one or a few of these services are performing inadequately, it would be possible to not invoke the inadequately performing services. This would of course depend on the QoS-contract with the customer, and especially the agreements on speed of service and accuracy of results.

So, in order to model variability in Web services, it will be necessary that the above types of variability can be captured in variation points of a service composition. However, only being able to model these variability types is not enough. It is possible, or even likely, there will be realization relations between variation points. For example, say the loan approval process should be able to support encryption. This is a high level choice that may have many consequences throughout the process definition, but this is not that interesting when configuring; only the high-level choice for encryption matters. Therefore, it must be possible to model these relations.

An illustration

As an illustration of the variation point types mentioned above, consider the order booking process (taken from [13]) shown in figure 3.1. This is not an actual process notation, but it is used as an example to show where variability in such a process could be modeled. In this process, orders are received by the process, after which a credit check is performed by contacting a credit check service. After that, the price for the order is requested from two different suppliers' services in parallel (placed next to each other in the diagram, with the flow splitting before and after) and the lowest price is determined. The supplier with the lowest price is selected, and the order fulfillment is then underway. Several other services are contacted and in the end the customer receives a reply.

The external services, such as the credit check service in the example, are only abstractly defined. It is therefore possible for them to have different implementations: they could be implemented by a specific service or by a composition of services. Figure 3.2 highlights one of the services in the order booking process. Each of these service interfaces defined in the process is a possible variation point in this way. This corresponds to the first type (service instance) in the enumeration above.

Another possible variation point in such a process is the interface of a certain service. It is possible there are several accepted "standards" for an interface to (or for the protocol used to access) a certain type of service, and it is therefore conceivable that they should all be supported. However, different interfaces usually have different parameter sets, so that would mean a part of the process logic (in particular the preparing of a message to that service) is influenced by the interface it is sent to (illustrated in figure 3.3). This corresponds directly to the second (service interface) and third (service parametrization) type in the enumeration above, although the second type overlaps with both this example and the previous.

It is also possible to view the composition of subservices as a variation point. For instance, in the process two price quote services are invoked in parallel after which their results are compared. This is not the only way the services could have been composed. Another possible composition would have been, for instance, a sequential invocation, because one of the suppliers is a preferred supplier, but it might not have all the items ordered by a customer in stock. This means that

in a process definition, it should be possible to designate a part wherein the activities performed are variable. Such a part of the service process is called a "service fragment". This is an example of the fourth type (system composition) of variation point. Figure 3.4 highlights the part of the process discussed.

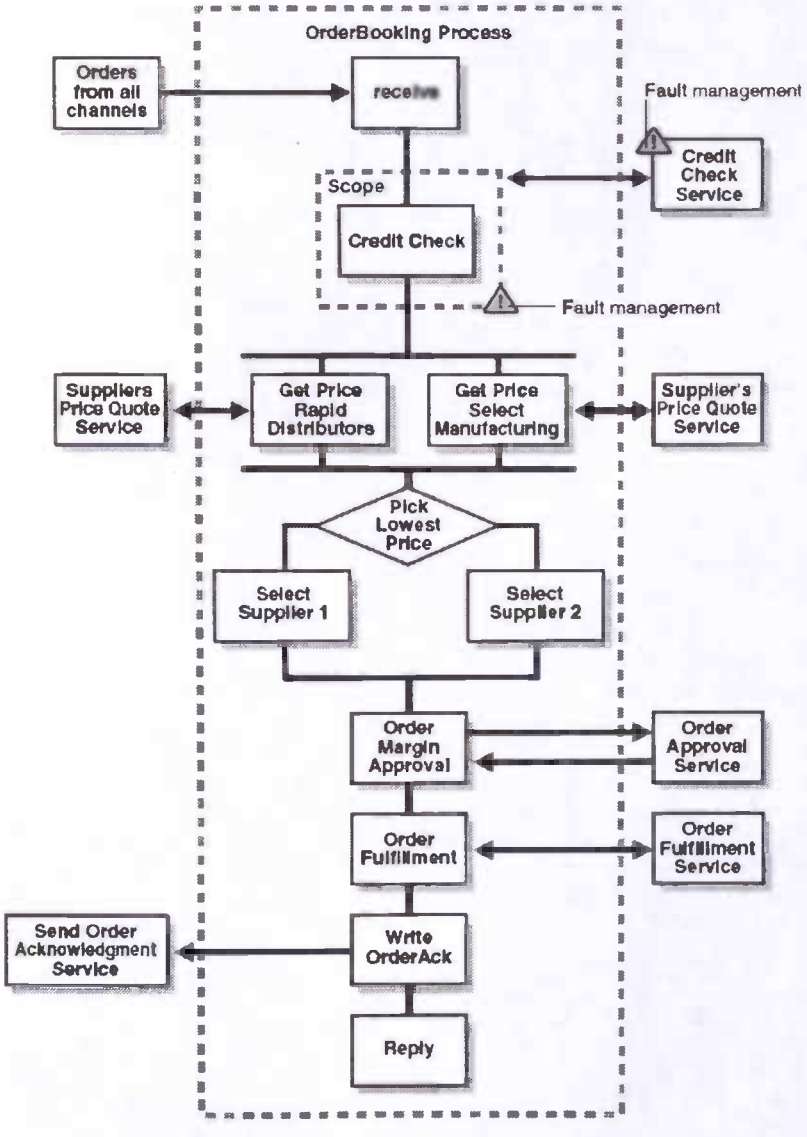

Figure 3.1.: (Non-standard) flow notation for an order booking process.

3.3. Run-time variability issues

As shown above, run-time variability offers a number of advantages. However, it is unfortunately not as simple to allow run-time variability as one may have liked. The most immediate issue is

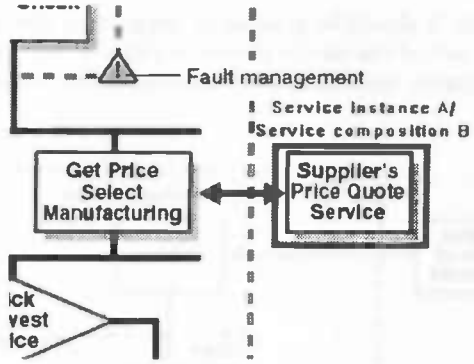

Figure 3.2.: A service is an abstract definition and can therefore have different implementations.

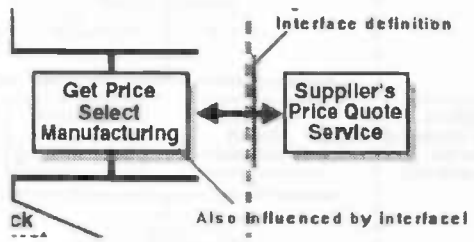

Figure 3.3.: The interface definition and the part of the process it influences when it is changed.

changing a running transaction or process in a (Web) service system. Once a process starts, it is possible that this process will not complete for several days, weeks, or even months, depending on the type and definition of the process. Altering the configuration of the system can, depending on what is altered, affect the executing process. If an invoked service in a system is interchanged with a different one with the same signature, this will not matter much, unless a process instance has just invoked the previously used service and is awaiting a reply. In other cases, such as altering the control flow in the system, it is likely that the running process cannot simply use the new system definition because of incompatibilities. Therefore, for the purposes of this thesis, run-time variability will be defined as variability between process invocations, while not affecting the structure of a running process. This ensures each invocation remains internally consistent.

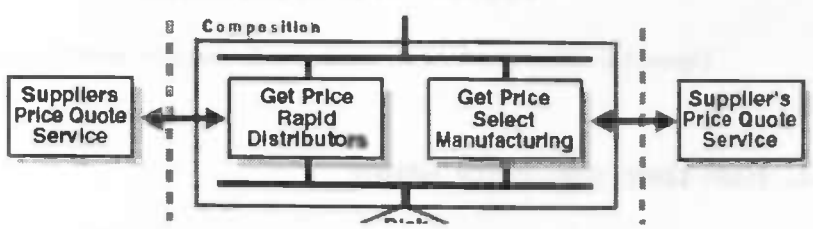

Figure 3.4.: A possible point in the process where a composition variation point can be located.

3.4. Research question

As shown above, there are several advantages to supporting run-time variability in service based systems. In order to take advantage of this explicit modeling, one will also need this variability to be manageable, at run-time and preferably externally. In order to provide a solution to this problem, the central question this thesis aims to answer, is "How can run-time variability and variability management be allowed, modeled and supported in a service-centric system, so it can quickly react or be adapted to changes in its dynamic environment?" This is however a rather broad question. Therefore, the following, more specific, questions will be answered in order to give an answer to the central question:

- Is it possible to make a service-centric system variable at run-time? How can processes be kept consistent internally?
- How can (run-time) variability be supported in a service centric system?
- How can the variability types as described in section 3.2 be captured in this modeling?
- How can abstraction (i.e., realization relation information) be supported?
- How can the variability information be modeled graphically in order to give a clear view on the variability in the system?
- Using modeling of variability information, how can a service-centric system's variability be made manageable externally and at run-time?

Throughout this thesis, an answer to each of these research questions is provided.

4. Extending BPEL

In order to be able to capture variability information in a service-centric system (as stated in section 3.4), one must adapt the way such a system is described. Service-centric systems are usually process-driven, and these processes are defined by a process description language. So, in order to capture variation point and dependency information in a service-centric system, the process description language used for such a system must to be adapted to allow variability information to be defined.

It is hard to do this for a process description language in general and therefore a solution is presented for WS-BPEL 2.0, one of the major standards for (executable) process description and definition languages, to show that it is possible for at least one.

4.1. Introduction to BPEL

BPEL was already mentioned in section 2.2.2. This subsection aims to give a more in-depth introduction to the language. The language is an XML-based flow language. It supports structured programming constructs such as if-statements, while-statements, sequence-statements (to execute statements in sequence) and flow-statements (to execute statements in parallel). Since it is focused on service-based business processes, it has support for sending messages natively, using WSDL to describe the messages and the services they are sent to.

The version of BPEL that is mostly used now is BPEL4WS 1.1[3], which was submitted by (among others) IBM, Microsoft and BEA to the OASIS to be standardized. However, a new standard, WS-BPEL 2.0, is currently in development, which is incompatible with BPEL4WS in several ways. The name change signifies the incompatibility. The incompatibilities include new statements as well as removed statements, and child elements replacing attributes for certain constructs. For an in-depth specification of WS-BPEL 2.0, see [11].

To illustrate what WS-BPEL looks like, figure 4.1 contains a simplified example process definition in WS-BPEL 2.0 for the process described in section 1.1. The process receives a request from a customer (`<receive>`) and determines whether the amount is smaller than 10,000 (`<transitionCondition>`). If so, an assessor service is queried (`<invoke>`) to determine the person's risk level. If this risk level is low, the answer `'yes'` is placed in a message (`<copy>`) and the customer is notified (`<reply>`). If the loan amount is 10,000 or more (`<transitionCondition`) or the customer's risk level is not `'low'`, an approval service is queried (`<invoke>`) and its answer is returned to the customer (`<reply>`). The process definition uses a `<flow>`-statement, which is a container for activities which will be executed in parallel. It has links and (possible) transition conditions for each link, making it similar to a state machine. However, this is not the only type of activity container: there is also, for example, a sequential container (`<sequence>`).

The rest of this section will discuss the extension to the BPEL language defined to allow variability to be modeled in business processes. The extension was named "VxBPEL" (Variability extended BPEL). The remainder of this chapter assumes that the reader has a basic understanding of the working of XML and XPath.

```
<process name="loanApprovalProcess">
  <flow>
    <receive partnerLink="customer" operation="request"
        variable="request" createInstance="yes">          <!-- A request is received     -->
      <sources>
        <source linkName="receive-to-assess">
          <transitionCondition>                           <!-- Two possible transitions: -->
            $request.amount &lt; 10000                    <!-- A low amount..           -->
          </transitionCondition>
        </source>
        <source linkName="receive-to-approval">
          <transitionCondition>
            $request.amount &gt;=10000                    <!-- or a high amount.        -->
          </transitionCondition>
        </source>
      </sources>
    </receive>

    <invoke partnerLink="assessor" operation="check"
      inputVariable="request" outputVariable="risk">      <!-- if the amount is low,    -->
      <targets>                                           <!-- determine the risk.      -->
        <target linkName="receive-to-assess"/>
      </targets>
      <sources>
        <source linkName="assess-to-setMessage">
          <transitionCondition>                           <!-- Two possible transitions: -->
            $risk.level='low'                             <!-- Low risk..               -->
          </transitionCondition>
        </source>
        <source linkName="assess-to-approval">
          <transitionCondition>
            $risk.level!='low'                            <!-- or not low risk.         -->
          </transitionCondition>
        </source>
      </sources>
    </invoke>

    <assign>
      <targets>
        <target linkName="assess-to-setMessage"/>
      </targets>
      <sources>
        <source linkName="setMessage-to-reply"/>
      </sources>
      <copy>                                              <!-- If both amount and risk  -->
        <from><expression>'yes'</expression></from>       <!-- were low, automatically  -->
        <to variable="approval" part="accept"/>           <!-- accept the loan request. -->
      </copy>
    </assign>

    <invoke partnerLink="approver" operation="approve"
       inputVariable="request" outputVariable="approval"> <!-- If amount was high or risk -->
      <targets>                                           <!-- wasn't low, invoke another -->
        <target linkName="receive-to-approval"/>          <!-- approval service.          -->
        <target linkName="assess-to-approval"/>
      </targets>
      <sources>
        <source linkName="approval-to-reply" />
      </sources>
    </invoke>

    <reply partnerLink="customer"                         <!-- The reply is either the  -->
       operation="request"                                <!-- automatic accept ('yes') -->
       variable="approval">                               <!-- or the answer from the   -->
      <targets>                                           <!-- external approver service.-->
        <target linkName="setMessage-to-reply"/>
        <target linkName="approval-to-reply"/>
      </targets>
    </reply>
  </flow>
</process>
```

Figure 4.1.: An example of a BPEL process definition

4.2. An extension to BPEL: VxBPEL

In order for BPEL to model variability, for which it has no native support, additional XML elements will have to be introduced as an extension to BPEL.

First of all, it will be necessary to choose how to extend BPEL to store the relevant variability information in a BPEL file. There are several options for this:

- Adding the information to a separate file, using an XML query language (such as XPath) to point to where the variation points should be inside the process definition.
- Adding the information in the BPEL file, but separately from the original BPEL elements, again using an XML query language to point to where the variation points should be in the process definition.
- Adding the information as extension elements inside the process definition itself, using a different namespace. This is also the recommended way to extend an XML format like BPEL.
- Adding the information inside the BPEL process as comments.

Conceptually, the first two options are the same. Adding information to either a different file or the same file, using XPath to point to specific elements, has no real influence on how the information is stored about the process definition. Therefore, these two options will be considered equal for the rest of this chapter.

While adding the information as comments is possible, one (conceptual) problem with that is that it would imply the information that is stored inside the comments is not relevant to the process itself. This is, of course, untrue, since variability information is very relevant to the process. This is the reason this option will not be considered further.

Two options now remain to model the variability information: separating the variability information from the process, or adding this information into the process definition itself. Section 4.3 briefly discusses two processes used as illustration for the BPEL extensions.[1] The next two subsections will show how variability information would be added to the process definition for each of the two modeling options there are left. Their (dis)advantages will be discussed in the subsection following.

4.3. Examples used as illustration

This section uses the example described in section 1.1 as a basis for two processes. The first (original) process uses the example without changes. The associated BPEL (BPEL4WS 1.1) code can be found in appendix B.1. A simplified BPEL (WS-BPEL 2.0) source of this problem can be seen in figure 4.1.

The second process is a variant of the original process. In this variant, the assessor is a little different. It requires the same information as the original process, and in addition asks for an extra part in the message; an example might be information procured from a local database that is relevant to the customer's request. The example used in this chapter is kept rather more simple: a notes field is added, with just the text "Request for approval". This is done to make sure the example does not get too complicated and too hard to understand. An UML-like flow diagram of the variant process is shown in figure 4.2.

[1] Because of the lack of support for WS-BPEL 2.0 in ActiveBPEL, as will be discussed in section 5.1, the BPEL code for these examples is written in BPEL4WS 1.1.

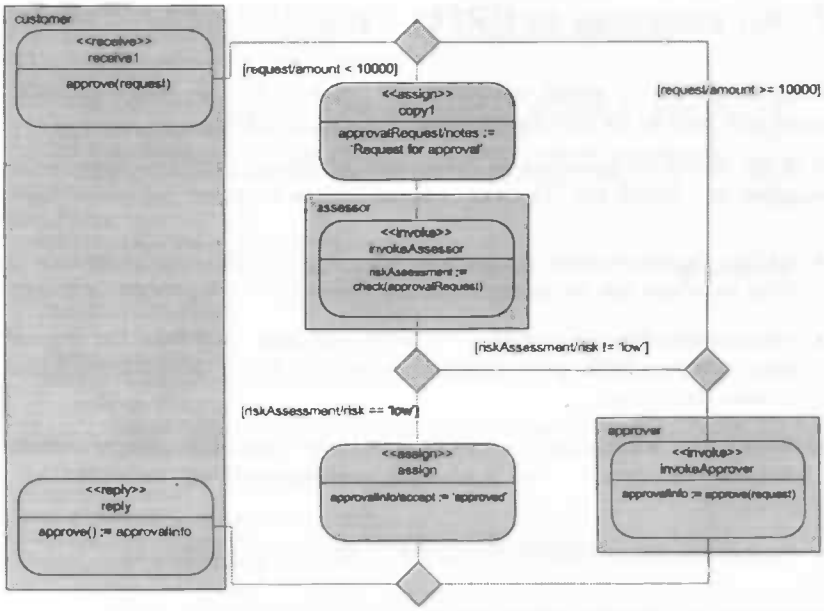

Figure 4.2.: The variant of the example process

4.4. Extending BPEL with variability information

4.4.1. Separate variability information

The first option to add variability information is to keep it separate from the process definition. This can either be done by adding it as a separate part in the process definition file, or by putting it in a separate file. We can add this information inline without affecting the original BPEL file by using a different namespace (`http://vxbpel.rug.org`, prefix `vxbpel` in the example). This ensures that standard interpreters of the BPEL file will ignore the elements added as well as everything contained in these elements (as defined in [11], 5.3).

When we wish to add the variability information discussed in the previous subsection as a separate part of the file, it will be necessary to indicate which points in the XML are definition are considered variation points. XPath provides the functionality needed to make this possible. The XPath expression `/process/flow/invoke[@name='invokeAssessor']` points to the `<invoke>` element that has the `name` attribute set to `invokeAssessor`.

It would also be possible to point to the same node with the expression `/process/flow/invoke[2]` (the second `<invoke>` element contained in the `<flow>` element), but this method has two disadvantages: firstly, the index will change when rearranging/reordering the nodes, invalidating the expression as a pointer to the variation point. Secondly, implementations of XPath exist that start counting at 0 as opposed to 1 as specified by the standard.

Pointing to nodes using the `name` attribute is a far more robust solution, especially because the `name` attribute can be added to every node in a BPEL document.

Using XPath, we can now define a `<vxbpel:VariationPoint>` that points to a specific element in the process definition as defined by its `path`-attribute. For each of these variation points, it is desirable to be able to indicate which variants exist for them. This is done by a sequence of `<vxbpel:Variant>` elements, enclosed by the container element `<vxbpel:Variants>`. Each of these variants has a name as indicated by the `name` attribute, and associated BPEL code to be

placed in the process definition, defined by the <vxbpel:BpelCode> element. The definition of a variation point is shown in the listing in figure 4.3.

```
<vxbpel:VariationPoint name="VP1" path="/process/flow/invoke[@name='invokeAssessor']">
  <vxbpel:Variants>
    <vxbpel:Variant name="default">
      <vxbpel:VPBpelCode>
        <invoke inputVariable="request" name="invokeAssessor" operation="check"
                outputVariable="riskAssessment" partnerLink="assessor"
                portType="asns:riskAssessmentPT">
          <target linkName="receive-to-assess"/>
          <source linkName="assess-to-setMessage"
            transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
          <source linkName="assess-to-approval"
            transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
        </invoke>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
    <vxbpel:Variant name="alternative1">
      <vxbpel:VPBpelCode>
        <sequence name="sequence">
          <target linkName="receive-to-sequence"/>
          <source linkName="sequence-to-setMessage"
            transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
          <source linkName="sequence-to-approval"
            transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
          <assign name="copy1">
            <copy>
              <from expression="'Request for approval'"/>
              <to part="notes" variable="approvalRequest"/>
            </copy>
          </assign>
          <invoke inputVariable="approvalRequest" name="invokeAssessor"
                  operation="check" outputVariable="riskAssessment"
                  partnerLink="assessor" portType="asns:riskAssessmentPT">
          </invoke>
        </sequence>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
  </vxbpel:Variants>
</vxbpel:VariationPoint>
```

Figure 4.3.: Definition of a variation point, separate from the process definition.

After defining variation points and the variants between which can be chosen, the last information needed to make the process configurable are the so-called "configurable variation points", <vxbpel:ConfigurableVariationPoint>. These are variation points at a higher level. Each of these configurable variation points also has variants, <vxbpel:Variant>, enclosed in the <vxbpel:Variants> element, and for each of these variants an element <vxbpel:RequiredConfiguration> exists, which indicate for each high level variant what lower-level variants need to be selected through a number of <vxbpel:VPChoice>s. In other words, these high-level variation points cover realization relations. The only variation points that should be actively selected are these, as then the lower-level variation points will automatically be set accordingly. To help the user (or a process that automates process reconfiguration) select the correct variant, information is added about the variation points and the variants in the <vxbpel:Rationale> and <vxbpel:VariantInfo> elements. If this information is formalized, automatic configuration is possible. An example definition of a configurable variation point is shown in the listing in figure 4.4.

For this separate approach, there should always be a default variant for each configurable variation point. This is necessary because in this notation, the BPEL process definition is in a default

```
<vxbpel:ConfigurableVariationPoints>
   <vxbpel:ConfigurableVariationPoint id="CVP1" defaultVariant="default">
      <vxbpel:Name>The name goes here.</vxbpel>
      <vxbpel:Rationale>
         <!-- Here information should be put that explains the goal of this VP
              and what consequences configuring this VP has.
              This information could be formal or informal. Formal information
              would allow autonomous or automatic configuration. -->
      </vxbpel:Rationale>
      <vxbpel:Variants>
         <vxbpel:Variant name="default">
            <!-- The name of the variant is now identical to the variant names of the
                 associated VPs. This is not assumed to always be the case. -->
            <vxbpel:VariantInfo>
               <!--  Information is put here that pertains only to this variant. -->
            </vxbpel:VariantInfo>
            <vxbpel:RequiredConfiguration>
               <vxbpel:VPChoices>
                  <vxbpel:VPChoice vpname="VP1" variant="default"/>
               </vxbpel:VPChoices>
            </vxbpel:RequiredConfiguration>
         </vxbpel:Variant>
         <vxbpel:Variant name="alternative1">
            <vxbpel:VariantInfo>No extra info.</vxbpel:VariantInfo>
            <vxbpel:RequiredConfiguration>
               <vxbpel:VPChoices>
                  <vxbpel:VPChoice vpname="VP1" variant="alternative1"/>
               </vxbpel:VPChoices>
            </vxbpel:RequiredConfiguration>
         </vxbpel:Variant>
      </vxbpel:Variants>
   </vxbpel:ConfigurableVariationPoint>
</vxbpel:ConfigurableVariationPoints>
```

Figure 4.4.: Definition of a configurable (high-level) variation point, including the realization relations.

configuration (i.e., the activities listed in the process definition) and this default configuration is recorded in both the process definition and the variation point information (indicated by the `defaultVariant` attribute for each configurable variation point). This is an unfortunate consequence when one wants to ensure that the process definition is still executable when the variability information is ignored, as it is then necessary to know which parts of the original process definition to replace or remove when selecting a variant other than the default.

A possible work-around is to make the process definition contain only `<empty>` elements at the places where a variation point is to be defined. This would however mean that the default executable process is rather trivial.

The full separate approach example can be found in appendix B.2.

4.4.2. Inline variability information

To include information in the BPEL process inline, BPEL elements are enclosed by elements from a different namespace (the `vxbpel` namespace in the example). In this way, it is possible to see which parts of the process are variable by looking at the definition. Unfortunately, this means the BPEL file can no longer be interpreted by a BPEL engine "as is", as the elements from another namespace, including their children, are ignored by an interpreter. However, a simple transformation using, for example, XSLT would solve this.

To indicate that a part of a BPEL process is a variation point, it is enclosed by a `<vxbpel:VariationPoint>` element. Variants defined for this variation point are listed within

such an element by a `<vxbpel:Variant>` element. The `<vxbpel:VPBpelCode>` element contains the BPEL elements associated with the enclosing variant. This is very similar to how variation points are defined in the separate approach as shown in figure 4.3.

The configurable variation points are defined in the same way as the separate approach, i.e. after the process definition; see figure 4.4.

The full inline approach example can be found in appendix B.3.

4.4.3. Advantages and disadvantages of both approaches

Obviously, neither approach is perfect. Both have advantages and disadvantages this section will discuss.

Separate approach

The first advantage of the separate approach is that the original process definition is not altered in any way. It can be executed by any BPEL engine that ignores tags from a different namespace, as defined in the BPEL4WS 1.1[3] and WS-BPEL 2.0[11] specification. Another advantage is that all the information related to variability is together in one place, presenting a good overview in theory. However, one can imagine that when a process has upwards of 20 variation points, these being defined in XML will not make it easy to read. On top of that, when one wants an overview of the variability, it is most likely a graphic modeling tool will be used to display the variability graphically.

A disadvantage is that indicating a node by XPath can be error-prone, and there are sometimes errors in implementations: there is one XPath implementation that starts indexing at 0, as opposed to 1 as defined in the XPath specification. Rearranging elements in the process might require a reevaluation of all the XPath expressions to make sure they are still correct. This problem can be avoided, however, by making sure all the elements have names, and enforce this in tooling used to define variable processes. Also, it is difficult for anyone looking at the process definition (more so in complex examples) to determine which parts are currently considered variation points. Another disadvantage is the need to duplicate code in case the process definition itself should be executable by any BPEL engine - which also makes maintaining these processes more difficult than necessary.

Inline approach

A big advantage of the inline approach is that the variability information is located inside the process definition, which makes defining a process as well as implementing a parser or reader for this variability information easier. Also, by looking at the process definition, it is significantly easier for a human to see the variability in the process. However, as mentioned in the discussion about the separate approach, one wanting an overview of the variability will most likely use or develop tooling for this.

A disadvantage is that extending BPEL like this makes new process definitions incompatible with the BPEL format and it will no longer be possible for standard engines to read the definition. However, this is also an advantage - if variability is explicitly modeled, it might not be desirable at all to be able to execute it regardless. Also, it is possible to transform the process using an XSLT to conform to the standard BPEL format once more, should one really need to execute the process on a standard BPEL engine.

4.4.4. Decision

In the previous sections, two ways of representing variability information in a BPEL file have been presented: defining this information separately, or defining this inline in the process definition. Both approaches have their advantages and disadvantages, as summed up in the previous subsection. It is still debatable, however, which of the approaches is actually the one to choose. Neither approach has (dis)advantages significant enough to completely discard it, and so either is viable.

I have decided to use the inline approach for this thesis, because:

- It does not require code duplication, which is error-prone.
- It is easier to implement and to define processes manually, as there obviously does not exist any tooling for defining processes with variability information.
- Most of all, it is less complicated when parsing, so less time-consuming.

4.5. Conformance to variability modeling requirements

Now that the extension elements to BPEL are presented, one might wonder how the different types of variation points as discussed in section 3.2 can be modeled. This is shown here.

The idea behind the VxBPEL extensions was to model variability generically, so each of the different types of variability could be captured in a uniform way. Rather than specifically allowing the types mentioned in section 3.2 alone, VxBPEL was designed to be able to model all these types in the same way and thus have more flexibility. For each of the types mentioned, this section briefly discusses how modeling each type is possible.

4.5.1. Service replacement

This actually covers both the first (replacing a service by one with the same interface) and second (replacing a service by one with a different interface) type. Although BPEL of itself allows services with identical interfaces to be bound at run-time, it is conceivable one wants to define explicitly which service to use for which configuration of the system. In that case, an extra partner link could be added for each variant, and each of these variants would call a different service. In VxBPEL:

```
<partnerLinks>
   <partnerLink myRole="..." name="..." partnerLinkType="..."/>
   <partnerLink name="service1" partnerLinkType="..." partnerRole="..."/>
   <partnerLink name="service2" partnerLinkType="..." partnerRole="..."/>
   <partnerLink name="service3" partnerLinkType="..." partnerRole="..."/>
</partnerLinks>
```

would become:

```
<partnerLinks>
   <partnerLink myRole="..." name="..." partnerLinkType="..."/>
   <partnerLink name="service1a" partnerLinkType="..." partnerRole="..."/>
   <partnerLink name="service1b" partnerLinkType="..." partnerRole="..."/>
   <partnerLink name="service2" partnerLinkType="..." partnerRole="..."/>
   <partnerLink name="service3" partnerLinkType="..." partnerRole="..."/>
</partnerLinks>
```

and a variation point could be modeled thus:

```
<vxbpel:VariationPoint name="VPService1">
   <vxbpel:Variants>
      <vxbpel:Variant name="Service1A">
         <invoke inputVariable="..." name="..." operation="..."
                 outputVariable="..." partnerLink="service1a" portType="..."/>
      </vxbpel:Variant>
      <vxbpel:Variant name="Service1B">
         <invoke inputVariable="..." name="..." operation="..."
                 outputVariable="..." partnerLink="service1b" portType="..."/>
      </vxbpel:Variant>
   </vxbpel:Variants>
</vxbpel:VariationPoint>
```

As the actual interface for a service is captured in the `portType` and `partnerLinkType`, one can see that modeling a variation point as such means that it is possible to define both `invoke` statements with different parameters, thus allowing both types of variability to be captured. As a sidenote, it is possible the input and output variable differ for both services, in which case the surrounding statements which prepare a message for sending will also need to be adapted.

4.5.2. Service parameters

Modeling this type is similar to how modeling a variation point offering a choice between two services is done. However, it is dependent on how the parameters for this service need to be set: either by altering the message sent to this service, or by first invoking a different operation of a service in order to set parameters for a next request. Either way, surrounding statements (be it an `invoke` statement to call a different operation or an `assign` statement to change the message contents) will need to be adapted. Suppose a service is normally called without setting parameters beforehand (i.e., use the default settings):

```
<invoke inputVariable="input" name="invokeService" operation="operation"
        outputVariable="output" partnerLink="serviceName" portType="..."/>
```

and one wants to be able to set parameters for a service first, by invoking an operation that sets the service parameters:

```
<sequence name="setserviceparams">
  <invoke inputVariable="params" name="invokeService" operation="setParams"
          outputVariable="setParamsOut" partnerLink="serviceName"
          portType="..."/>
  <invoke inputVariable="input" name="invokeService" operation="operation"
          outputVariable="output" partnerLink="serviceName"
          portType="..."/>
</sequence>
```

one would then model a variation point thus:

```
<vxbpel:VariationPoint name="confService">
   <vxbpel:Variants>
      <vxbpel:Variant name="noParams">
         <invoke inputVariable="input" name="invokeService"
```

```
                operation="operation" outputVariable="output"
                partnerLink="serviceName" portType="..."/>
      </vxbpel:Variant>
      <vxbpel:Variant name="setParams">
         <sequence name="setserviceparams">
            <invoke inputVariable="params" name="invokeService"
                    operation="setParams" outputVariable="setParamsOut"
                    partnerLink="serviceName" portType="..."/>
            <invoke inputVariable="input" name="invokeService"
                    operation="operation" outputVariable="output"
                    partnerLink="serviceName" portType="..."/>
         </sequence>
      </vxbpel:Variant>
   </vxbpel:Variants>
</vxbpel:VariationPoint>
```

If one sets parameters on a service by invoking a different operation altogether or altering the type of message sent, it can be modeled in the same way as in the previous section.

4.5.3. System composition

System composition, or more in general, service fragments, can be modeled with VxBPEL as well. Suppose we have the following composition or fragment, which uses the parallel execution container `flow`:

```
<flow name="flow1">
   <links>
      <link name="A"/>
      <link name="B"/>
   </links>
   <invoke inputVariable="..." name="..." operation="..."
           outputVariable="..." partnerLink="..." portType="...">
      <sources>
         <source linkName="A">
            <transitionCondition>...</transitionCondition>
         </source>
         <source linkName="B">
            <transitionCondition>...</transitionCondition>
         </source>
      </sources>
   </invoke>
   <invoke inputVariable="..." name="..." operation="..."
           outputVariable="..." partnerLink="..." portType="...">
      <targets>
         <target linkName="A"/>
      </targets>
   </invoke>
   <invoke inputVariable="..." name="..." operation="..."
           outputVariable="..." partnerLink="..." portType="...">
      <targets>
         <target linkName="B"/>
      </targets>
   </invoke>
```

```
</flow>
```

and we want to define a variant with a sequential container, sequence:

```
<sequence name="sequence1">
   <invoke inputVariable="..." name="..." operation="..."
           outputVariable="..." partnerLink="..." portType="..."/>
   <if name="...">
      <condition>...</condition>
      <then>
         <invoke inputVariable="..." name="..." operation="..."
                 outputVariable="..." partnerLink="..." portType="..."/>
      <elseif>
         <condition>...</condition>
         <invoke inputVariable="..." name="..." operation="..."
                 outputVariable="..." partnerLink="..." portType="..."/>
      </elseif>
   </if>
</sequence>
```

one can then define a variation point with these fragments as variants:

```
<vxbpel:VariationPoint name="fragmentExample">
   <vxbpel:Variants>
      <vxbpel:Variant name="flow">
         <flow name="flow1">
            <!-- here goes the rest of the flow code... -->
         </flow>
      </vxbpel:Variant>
      <vxbpel:Variant name="sequence">
         <sequence name="sequence1">
            <!-- here goes the rest of the sequence code... -->
         </sequence>
      </vxbpel:Variant>
   </vxbpel:Variants>
</vxbpel:VariationPoint>
```

thus allowing changing service fragments or service composition.

5. ActiveBPEL, COVAMOF and VxBPEL

In order to validate that VxBPEL can indeed allow processes to be made variable, some testing will be necessary. This is why an existing engine, ActiveBPEL, has been adapted to allow VxBPEL processes to be run.

Also, in order to be able to graphically model variation points in a VxBPEL process and allow users to get an overview of variation points, variants and realization, tooling will be needed that can interpret VxBPEL variability information and can present this to the user. For this purpose, a plug-in for the variability modeling framework COVAMOF was developed.

This chapter discusses both the adaptation of the ActiveBPEL engine as well as the implementation of the COVAMOF plug-in.

5.1. Extending the ActiveBPEL engine

ActiveBPEL[7] is one of the most widely used BPEL engines of today. It is developed by Active Endpoints and is a commercial-grade open-source BPEL engine, written in Java and released under the GPL. It does not have support for WS-BPEL 2.0 yet, as this standard is not complete, but can interpret any standard BPEL4WS 1.1 process definition, as it fully conforms to this standard. The ActiveBPEL engine runs in any standard servlet container such as Apache Tomcat.

ActiveBPEL was selected as the engine to extend and adapt to interpret VxBPEL, because it is open-source and widely used. The next sections describe ActiveBPEL's architecture and the modifications made to the engine to allow VxBPEL processes to be executed.

5.1.1. ActiveBPEL's architecture

In order to be able to describe the changes made to ActiveBPEL, this section will first describe the relevant parts of ActiveBPEL's architecture. This description is by no means complete, but will contain the parts which have proven to be relevant to the adaptation of ActiveBPEL to allow VxBPEL files to be executed.

First of all, it is important to note that ActiveBPEL makes extensive use of the Composite and Visitor patterns[8]. The reader unfamiliar with these patterns can find an brief overview of them in appendix D.

As said before, a Web-service-centric system is process-driven. The process is defined in a process description language (in ActiveBPEL's case in BPEL). When such a process definition is deployed to the ActiveBPEL engine, the engine reads in the elements, or activities (such as `flow` and `invoke`), from such a BPEL process description file and stores them in so-called Activity Definition objects. These objects encapsulate all the properties defined in the process definition and are essentially a "blueprint" for process instantiation. They contain all the information needed to be able to execute the activities defined in the process description.

When a request to a deployed process is received by the ActiveBPEL engine, the set of activity definitions that is associated to this process is located. The engine then instantiates this set of activity definitions to a set of Activity Implementation objects. The implementation objects allow

the ActiveBPEL engine to execute them by calling their `execute` method. The instantiation is done through a Visitor object visiting all the activity definition objects and instantiating the associated implementation objects. When a process starts executing, the activities that need to be executed are added to an execution queue, allowing the process' execution to be easily suspended and resumed.

The way processes are read in from definition files needs a more in-depth explanation. Activities are read in by Reader objects that are specific to this type of activity. A reader takes care of initialising the definition object which is associated to the activity and other tasks such as linking it to its parent element's definition object. ActiveBPEL maintains a set of two repositories for these reader objects, called registries. One of these registries stores readers for BPEL elements, the other stores readers for extension elements, elements which are not part of the ActiveBPEL language. Which reader is retrieved from the registry for a certain element is determined by the element that is the parent of the current element.

The engine reads in the process definition, and retrieves readers for each element in this definition from either of the two registries ActiveBPEL maintains. These readers instantiate definition objects, which the engine consequently stores and the engine will then await invocation of this process.

5.1.2. Modification of ActiveBPEL

VxBPEL introduces new elements that need to be read by the ActiveBPEL engine. As such, the ActiveBPEL engine will need to be adapted in order to allow the following elements to be read in and stored:

- New elements such as `VariationPoint` and `Variants` in the process definition itself
- New elements such as `VariabilityConfigurationInformation` outside of main control flow of the process definition

It is desirable that, in whatever way these elements are stored, there is no need for fundamental changes in the ActiveBPEL engine, as it is under continuous development. When the WS-BPEL 2.0 standard is finished and released, it is highly like that ActiveBPEL will be made compatible with it and it is possible the engine's source code changes significantly to accommodate for incompatibilities.

Fortunately, because of the use of the registries mentioned in the previous section, this is not hard to achieve. The registries are flexible enough to allow us to add a Reader that can handle the new elements, and because it is associated to a specific parent element, one can restrict the places where these new elements are valid.

At first glance, it seems logical to use the extension registry to store the new elements in. However, due to various reasons discussed in section 5.2, it is actually more desirable to adapt the BPEL registry to read in the elements inside the process definition itself, and adapt the extension registry for elements added outside of the main control flow of the process.

Now, how can code that is external to the original BPEL engine code be executed without altering some files distributed by ActiveBPEL? ActiveBPEL allows configuration of several interesting properties, such as the class which is the implementation of the BPEL engine and as such the class that will be instantiated as the engine. As ActiveBPEL is itself deployed to a servlet container such as Apache Tomcat, one deploys this configuration file together with the ActiveBPEL code, and the correct engine object will automatically be instantiated after deployment by a "loader" class.

Combining this with the flexibility the registries give us, altering the registries BPEL maintains is fairly straightforward. All that needs to be done is extending the original BPEL engine object and in this implementation alter the registry which is initialised by the original BPEL code.

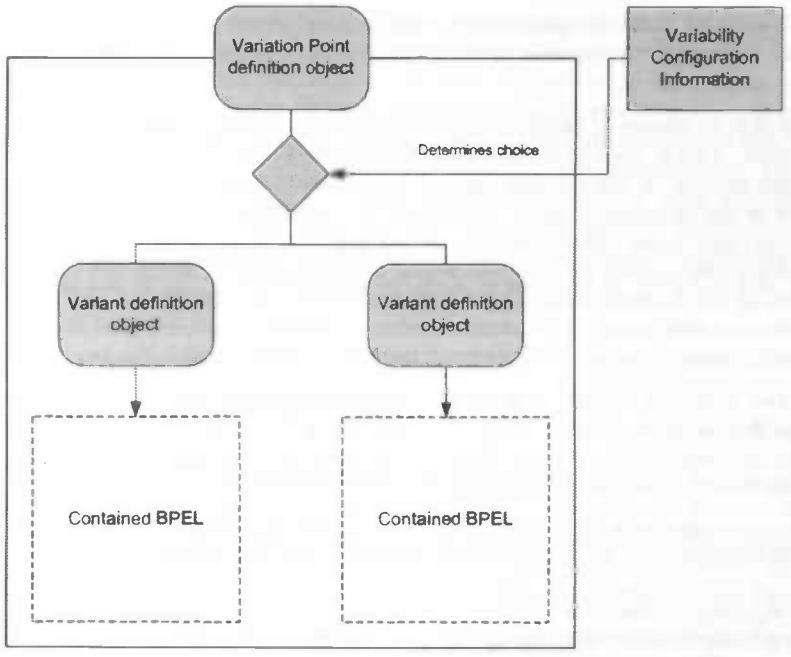

Figure 5.1.: The variation point and variant definition object structure.

5.1.3. Reading in and executing VxBPEL

In order to allow VxBPEL to be read in and executed, a new type of definition object will be added to the activity definition registry. This definition object is the Variation Point definition object. Using the fact that instantiation from definition objects to implementation objects is done by a Visitor visiting the definition object structure, the idea is the following. Basically, the Variation Point definition object is transparent to the Visitor that instantiates the process, by automatically forwarding requests to the BPEL element which is encapsulated by the currently configured Variant for this variation point. This way, there is no trace of the variability in the executable copy, the activity implementation objects, of the process definition and processes can be executed as if no variability exists, thus ensuring that process consistency is maintained. A graphic representation of the variation point/variant structure is shown in figure 5.1.

To make this possible, first several new readers and definition objects are defined. A reader is defined for each of the in-process extension elements: `VariationPoint`, `Variants`, `Variant` and `BpelCode`. These readers are stored in `VariationPointReader.java`, `VariantsReader.java`, `VariantReader.java`, `VariantReader.java` and `BpelCodeReader.java`. The readers are added to the BPEL registry, with all the BPEL elements that can contain variation points as possible parents. New definition objects are only defined for `VariationPoint`, `Variants` and `Variant` in respectively `ActivityVariationPointDef.java`, `VariantContainerDef.java` and `Activity-VariantDef.java`. The variation point definition object encapsulates the `VariantContainerDef` object which is the container for `Variants`. The variant definition objects contain a single activity (such as `invoke`) or activity container (such as `flow`). When a request from a visitor (which always happens through the `accept()` method) reaches a variation point definition object, it automatically forwards (based on configuration) the request to the activity or activity container in the correct, i.e. currently configured, variant definition object.

The variation points are defined as new activity definition objects due to use of inheritance, and as such can be successfully executed when contained by container activities for one or multiple

activities (e.g., <flow>, <sequence>, <while> and the case and otherwise structures that are part of a <switch> structure), when correctly added to the BPEL registry as described in section 5.1.1.

To be able to choose a variant based on the current configuration, this configuration will need to be stored. This is done by writing a reader to store the VariabilityConfigurationInformation element present in the process, defined in VarConfInfoExtensionReader.java. This reader is added to the extension registry and stores the variability information in a static class, defined in VarConfInfo.java. This class provides a mapping from a process' identifier (its name) to the variability information for this process and provides methods to retrieve configuration information, as well as set a configuration for a certain process. The reader for the variability configuration information also reads in the definitions of configurable variation points and stores this information in configurable variation point objects, defined in ConfigurableVariationPoint.java.

The new VxBPEL registry is defined in VxBpelRegistry.java and VxBpelDefReaderRegistry.java. This registry is loaded in by the VxBpelEngine object, defined in VariableBpelEngine.java. Refer to appendix E to see the package structure of VxBPEL and the file locations in the ActiveBPEL project.

In order to make sure the new engine is loaded after deployment, the following needed to be added to the configuration file which deploys together with the engine:

```
<!-- BpelEngine Impl -->
<entry name="EngineImpl" value="org.rug.vxbpel.server.engine.VariableBpelEngine"/>
```

5.2. Experiences and issues found during implementation

During implementation, several issues came up. First of all, after trying to define a reader for the new elements contained in the process, it turned out that placing these in the extension registry made it impossible to place the definition objects in the process' definition object structure and to read contained BPEL activities. This was due to the branch in the codepath that is followed in case a Reader object is taken from the extension registry. As both access to the object definition and readers for contained BPEL was necessary in order for the implementation to work, I decided to move these readers to the registry for BPEL elements.

The second implementation issue that turned up was executing a <flow> statement. Activities inside a flow statement are connected via <link> elements. Before a process is deployed, a validation is done for each flow statement to check that the links are correctly defined. However, the validation process is based on several assumptions that were true in the case of normal BPEL but could be false in case of a valid VxBPEL definition. For instance, the validation process does not allow links to be defined that are not used. This is a good thing for normal BPEL, as it makes sure there are no "loose links". However, the validation is done by a Visitor, and even though in case of a VxBPEL definition, it is possible all links are used, it is not possible for the validation process to determine that a link is used in a different variant due to the transparancy design of variation points. Allowing links to be unused required a small adjustment to the AeLinkValidator.java file.

For the <sequence> statement, a third issue emerged. During checking of which activities are allowed to create a new process, checking is done on a sequence statement that internally does not use a Visitor and can therefore be exposed to a VxBPEL Variation Point element, which requires special processing. This was circumvented by manually retrieving the configured variant from a variation point to be used in further processing. This required an adjustment to AeCheckStartActivityVisitor.java.

The validations done by ActiveBPEL before deploying a process caused a fourth issue. This time the problem emerged because several validations and other operations needed for execution, such

as setting the parent activity for each activity, are done before the process is actually deployed and externally accessible. When a variable process is deployed, these are only done at deployment time for the default configuration. This means that processes did not work after being reconfigured. The problem was fairly easy to fix, as all that needed to be done was re-run the validation on a process after its configuration was altered, but this needed a work-around. This was because the ActiveBPEL project is divided into several subprojects which are compiled in order. The validation is started from a different project from which the configurable variation point objects are in and as such, a configurable variation point could not start the validation. The work-around consisted of storing several objects, instantiated outside of the configurable variation point objects' project and needed for validation, in the configurable variation point objects. This required a change to `AeDeploymentValidator.java`.

5.3. Variability management using JMX

In section 3.4, one of the research questions was "Using modeling of variability information, how can a service-centric system's variability be made manageable externally and at run-time?". Using our newly defined objects, this is fairly easy to achieve by using JMX. JMX stands for Java Management eXtensions and is a way to explicitly expose functionality of objects to be able to monitor or manage them. Manageable objects are called MBean (for Manageable Bean).

To make an object manageable through JMX, several things need to be done. First of all, an MBean server needs to be created. Then, for each object you wish to be able to manage, you need to register it with the MBean server. These objects need to implement an interface with a name ending in "MBean" which specifies the operations that are available for management. After a server is created and objects implementing these special interfaces are registered, it is possible to connect to the MBean server and manage the registered objects.

The first step is to make sure an MBean server is created. This can be fairly easily done by changing the Tomcat configuration so it starts with JMX compatibility. In the version of Tomcat used, 5.0.28, and the JDK used, 1.4.x, JMX is not readily supported and therefore MX4J was used, an open source JMX implementation. After adding the MX4J jar to the classpath, an MBean server can be automatically started together with Tomcat by setting the following commandline options for Tomcat:

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=8081
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

This makes sure an MBean server is started on port 8081 without SSL and authentication enabled.

MX4J also ships with a useful tool called "MX4J/Http Adaptor" that enables a user to access manageable beans through a web browser. This tool can be enabled through Tomcat by installing a jar-file shipped with MX4J, `mx4j-tools.jar`, to the library path of Tomcat and altering Tomcat's configuration. After adding the following lines to `server.xml`:

```
<Connector port="${AJP.PORT}" handler.list="mx"
           mx.enabled="true" mx.httpHost="localhost"
           mx.httpPort="8082"
           protocol="AJP/1.3" />
```

the HTTP Adaptor tool is available through HTTP port 8082. The result, showing (a few of the) manageable beans available for Tomcat, can be seen in figure 5.2.

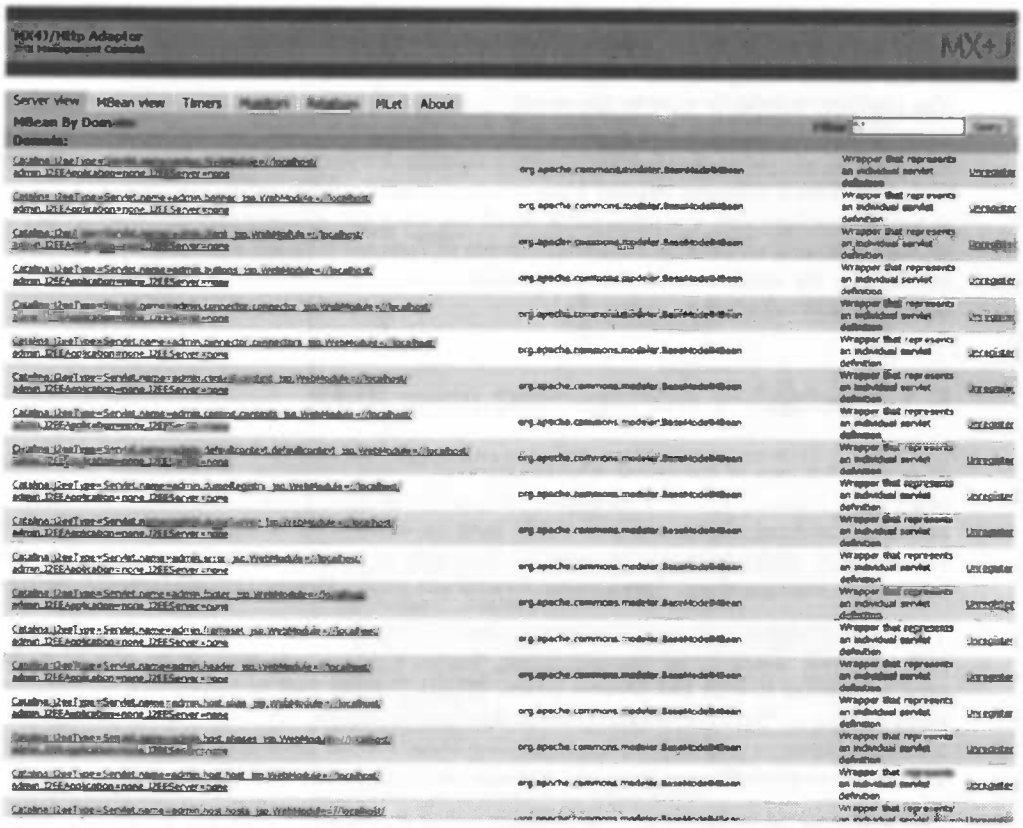

Figure 5.2.: The MX4J/Http Adaptor tool showing several of Tomcat's manageable beans.

All that remains now is adapting the objects we wish to make manageable and registering these with the MBean server. Because the high level variation points, the configurable variation points, should be manageable, these were adapted to implement an interface defined in `ConfigurableVariationPointMBean.java`, defining operations to show properties such as Rationale from the variation point, to request possible variant names and to set the current variant. Also, in the constructor of the configurable variation point objects, the object registers itself with the MBean server.

This enables a user to see which variation points are available through the JMX console, and to set a configuration for each variation point. The next section will show screenshots of how this works.

5.4. Testing the implementation

An example will show that, with the implementation of VxBPEL in ActiveBPEL and management via JMX, it is indeed possible to deploy and run a VxBPEL process and manage it at run-time. For this purpose, a webtool called ActiveBPEL Administration which ships with ActiveBPEL was used. It is a webtool that allows users to see the structure of deployed and executed processes from a browser application. By invoking the process, altering the configuration, and invoking the process again, it is possible to see differences in execution between these processes.

ActiveBPEL expects a BPEL file to be deployed as part of a set of files needed for execution. The files are stored together in a zip-archive which ActiveBPEL calls BPRs. Because there is rather a lot of extra information needed besides the pure BPEL in order to run a process (WSDL definitions, external service definitions and more), the example used for testing was derived from a sample for the ActiveBPEL engine published by ActiveBPEL itself. The version of the ActiveBPEL engine which was adapted, 2.0.0_1, had several examples available for download, one of which being a loan approval process. This sample, like all others, had all the files needed for deployment to and execution by the ActiveBPEL engine, as well as an ant build task for building and deploying the example.

The example used for testing is described below.

5.4.1. The example used for testing the implementation

The process used is a variation to the one mentioned in section 1.1. The original process is the same. A variant to this process is defined that, instead of a normal message, receives an encrypted message and will let a decryption service decrypt the request before processing it. The full text of this example can be found in appendix C.1.

Figure 5.3 shows what can be seen in the Administration tool after deploying this sample to the ActiveBPEL engine. On the left, a tree of the information stored in the process can be seen. On the right, a graphical representation of the process is shown, with information about the currently selected element at the bottom. Altering the configuration can be done by using the MX4J/HTTP Adaptor tool. What is visible of a configurable variation point is shown in figure 5.4. Clicking the "View Array" link in this figure gives us an overview of possible variant names that can be configured, as shown in figure 5.5. After setting the configured variant name to `encrypted`, we can look at the process definition again in the Administration tool, showing what is seen in figure 5.6.

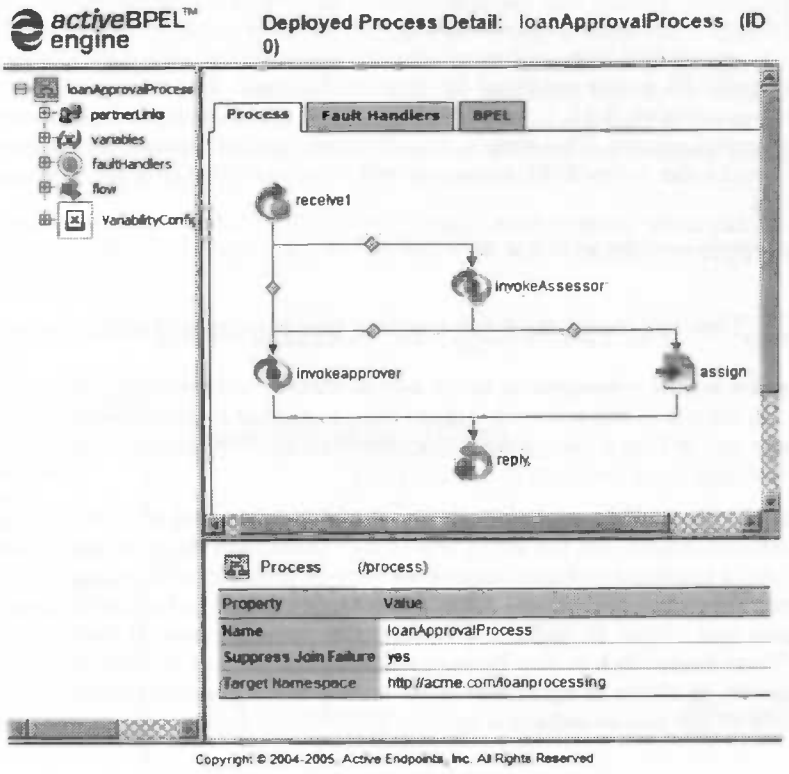

Figure 5.3.: The standard configuration.

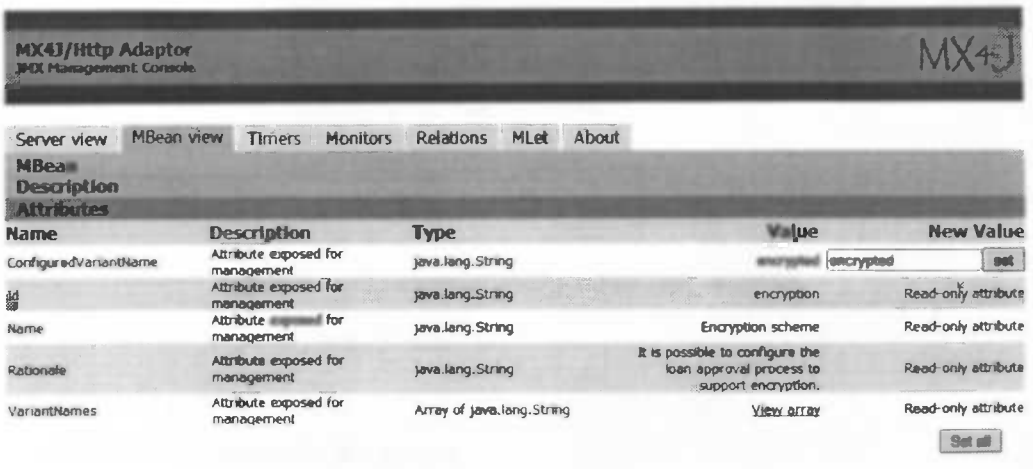

Figure 5.4.: The JMX HTTP Adaptor webpage where the configuration can be altered.

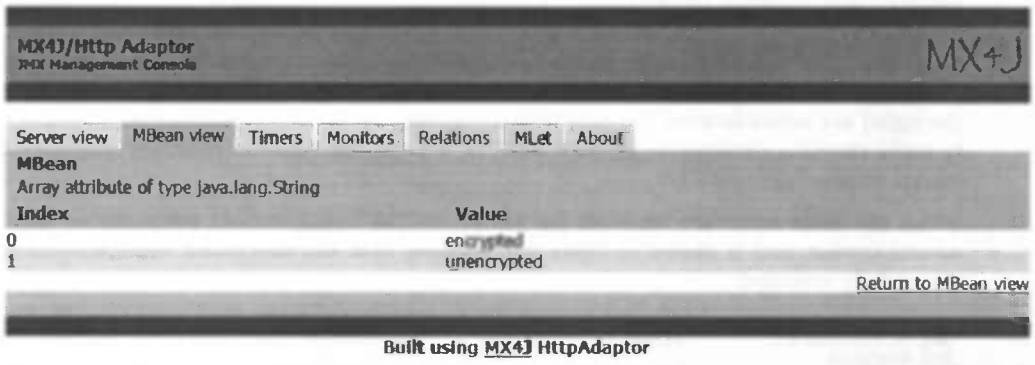

Figure 5.5.: Viewing the possible variants for a (high level, i.e. configurable) variation point.

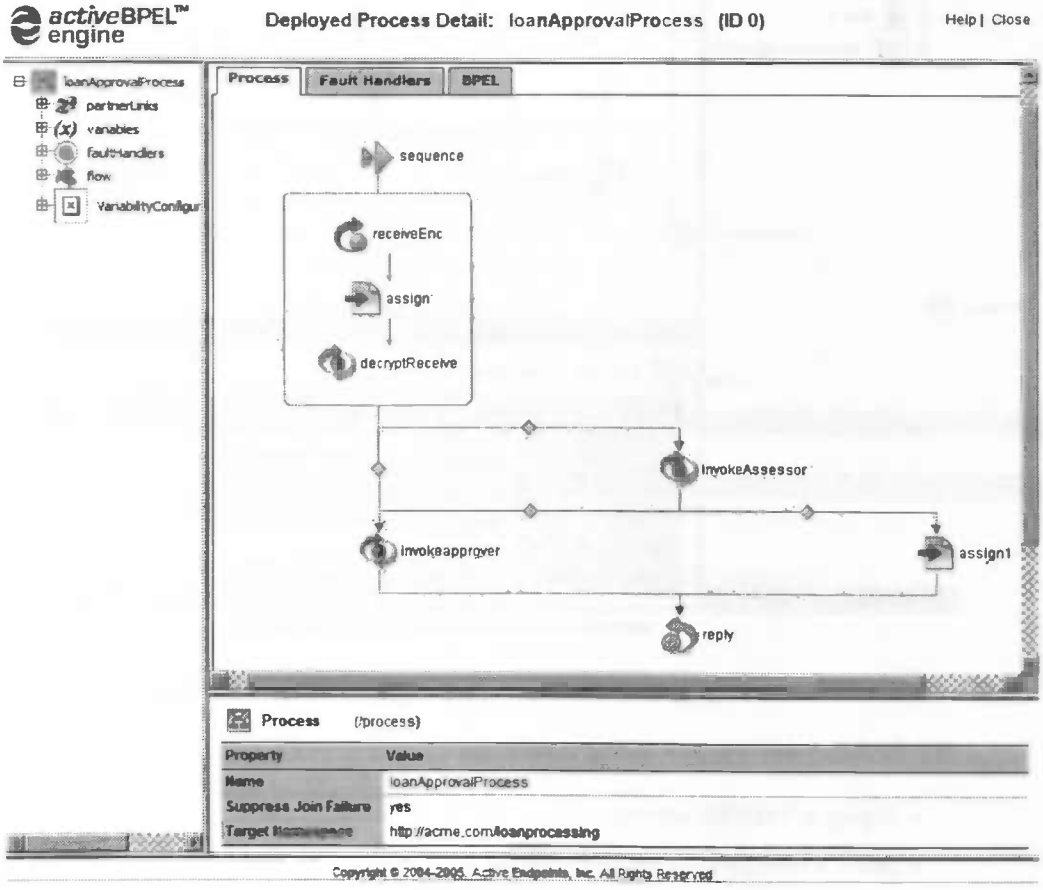

Figure 5.6.: The encryption configuration.

After the configuration is set to the standard configuration (i.e., no encryption) and the client application is executed, the details of the executed process can be viewed in the administration tool as shown in figure 5.7. This view is exactly the same as viewing a deployed process' details, but in the graphic representation of the process checkmarks, crosses, a symbol for a non-executed activity (a striked-through circle) and a symbol for a currently executing activity (a triangle pointing to the right) are added to show the execution's progress. As there is a path of activities from receive to reply showing checkmarks on each activity, we can see that the standard configuration was indeed successfully executed.

When the same steps are repeated for the encryption configuration, what can be seen in the administration tool is shown in figure 5.8, showing that the encryption configuration was also successfully executed.

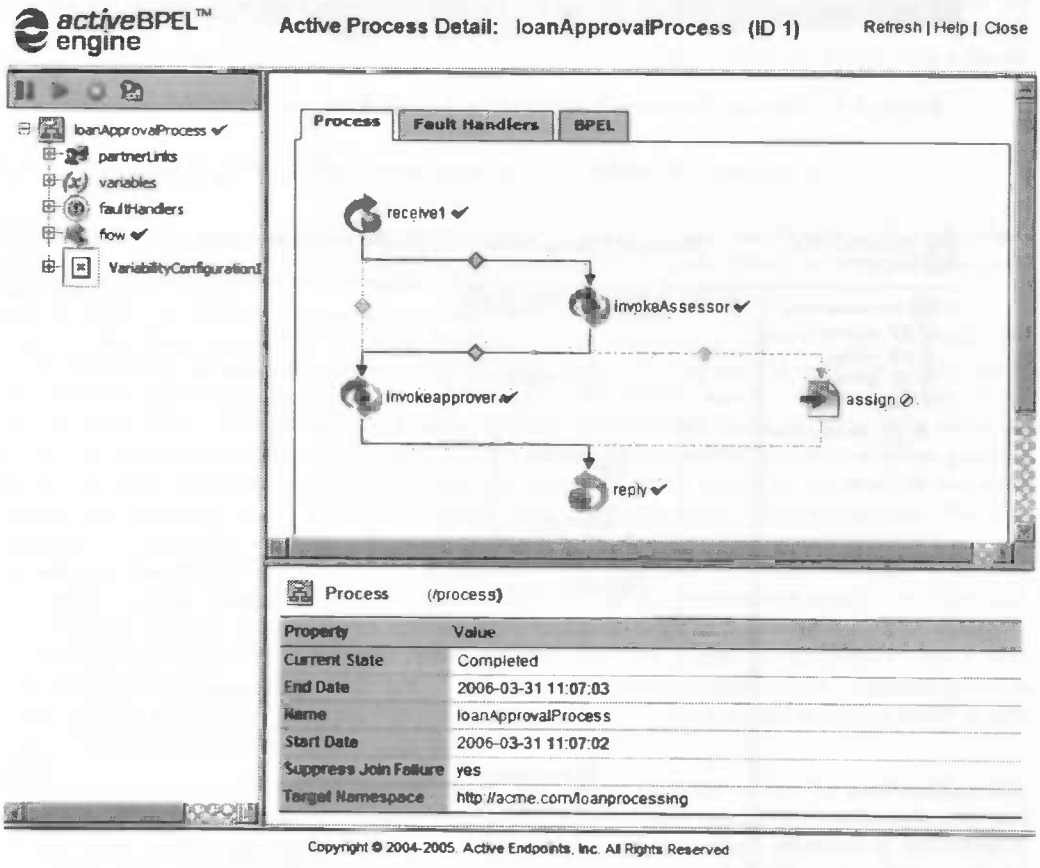

Figure 5.7.: The standard configuration after execution.

This demonstrates it is now indeed possible to:

- Deploy a VxBPEL process.
- Select a configuration for this process.
- Execute any of the variants for this process.

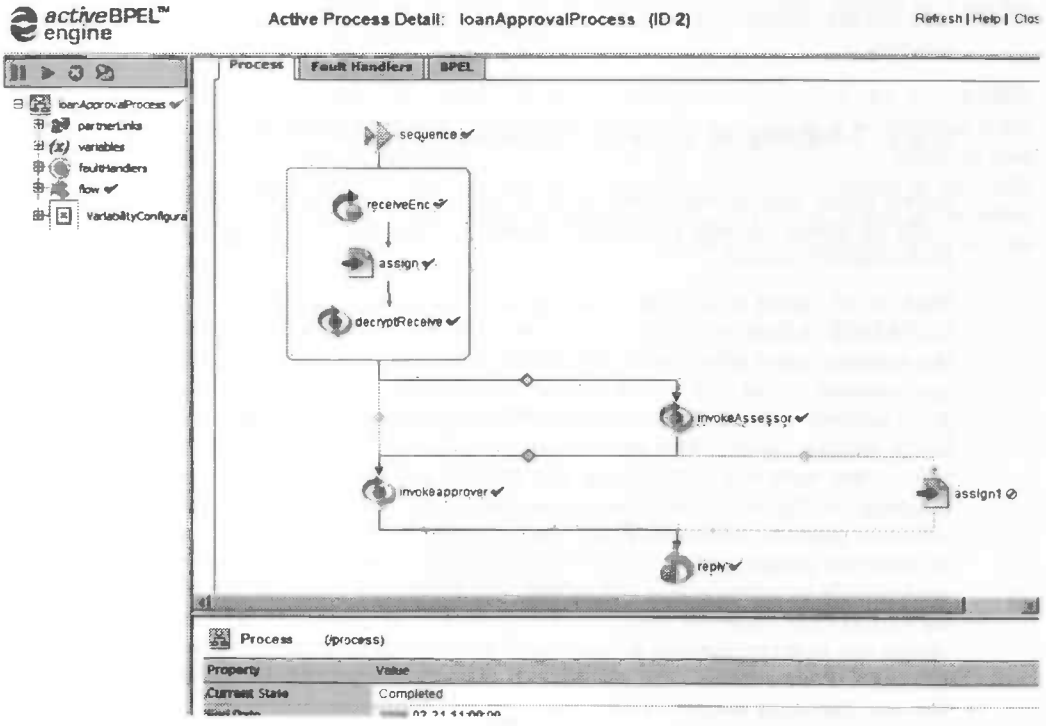

Figure 5.8.: The encryption configuration after execution.

5.5. COVAMOF and VxBPEL

When a complicated VxBPEL process contains many variation points and realizations, one can imagine that it becomes hard or impossible to have an overview of the variability by looking at the process definition alone, even with the separate `VariabilityConfigurationInfo`.

The COVAMOF framework and tooling offers facilities which would provide a better way to display information about variability. It can model the variation points and variants at different levels and can also model the realizations between them (i.e. the variation point choices modeled in the `VariabilityConfigurationInfo`). These realizations can be modeled by rules, enabling a user of the COVAMOF tooling to select a certain high-level variant, and see its impact immediately.

To make COVAMOF be able to work with VxBPEL files, a plug-in needed to be written for COVAMOF which is able to parse (relevant parts of) VxBPEL files and supplies the COVAMOF with the information the framework needs to model the variability. COVAMOF calls this a model provider.

5.5.1. Mapping of VxBPEL concepts to COVAMOF concepts

Before being able to implement a model provider, a mapping will need to be defined between VxBPEL constructs and COVAMOF concepts, which did not prove difficult as variability is modeled similarly in both.

First of all, there is a direct mapping between variation points in VxBPEL and COVAMOF. COVAMOF defines variation points at several levels and as having several subtypes. Only one of the variation point subtypes in COVAMOF is present in VxBPEL: variant variation points. These are variation points which have several variants that can be chosen and this is the only variation point concept currently present in VxBPEL. COVAMOF defines variation points at three possible levels, features, architecture and components (or none). VxBPEL only has two levels of variation points, the variation points inside the process and the configurable variation points. For this mapping, configurable variation points are (arbitrarily) defined to map to the architecture level variation points in COVAMOF and the in-process variation points are mapped to the lowest level of variation points possible in COVAMOF (being those at the components level). The fact that configurable variation points can be seen as configurable features in a process is a good reason to let configurable variation points map to feature level variation points in COVAMOF, but I have chosen not to do so, because it is clearer if both types of variation points map to adjacent levels in COVAMOF's modeling. The mapping of variants in VxBPEL to variants in COVAMOF is now also automatically defined.

What remains to be mapped are the variation point choices that are defined for each configurable variation point. As discussed in chapter 4, these choices actually model realization relations and as such map directly to these relations in COVAMOF. The realization relations in COVAMOF have associated rules that are evaluated to determine what happens for each realization relation if variation points are configured. These rules can be very complex in COVAMOF but for VxBPEL they only need to be fairly simple: a rule needs to define that a choice for a high-level variant dictates several choices on a lower level. Such a rule consists of a condition (being a certain variant chosen for a high-level variation point for VxBPEL) and several "bindings", that define which variant is set for which variation point.

5.5.2. Parsing of VxBPEL

If we want COVAMOF to only model the variability present in a VxBPEL process, the relevant information in VxBPEL files is:

- Variation points and variants in the process definition.
- Configurable variation points and their variants.
- Variation point choices associated with configurable variation points.

Basically, the COVAMOF plug-in will need to do the following:

- Parse the information in in-process variation points and store these as variation points and variants at the Components level.
- Parse the information in configurable variation points and store these variation points and variants at the Architecture level.
- Parse the required configuration for each configurable variation points' variant and storing the appropriate realization relations and rules.

When the plug-in is implemented and placed in the installation directory of the COVAMOF framework, we can now browse the variability in a VxBPEL process in a number of different ways, as illustrated in the screenshots in figures 5.9, 5.10, 5.11 and 5.12 (these figures use symbols used in COVAMOF as described in [16]). The VxBpel process used in these screenshots is the same as used in section 5.4.1 and its full text is given in appendix C.1. Note that names of variation points on these screenshots differ slightly from those in the example given before, as items are not allowed to have identical names in COVAMOF.

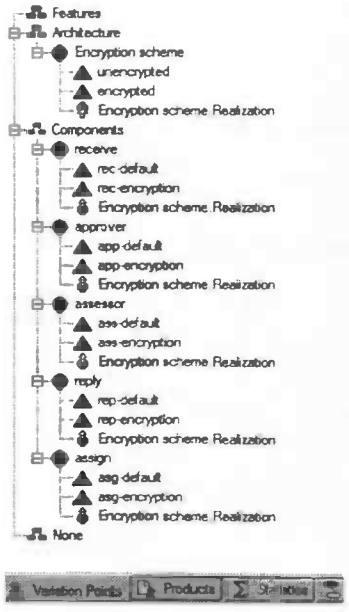

Figure 5.9.: The variation point view in COVAMOF, showing variation points on several layers.

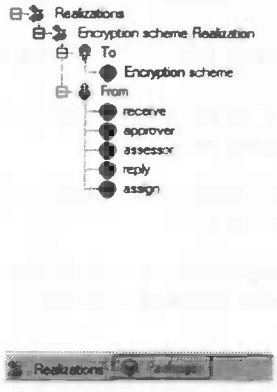

Figure 5.10.: Realization view in COVAMOF.

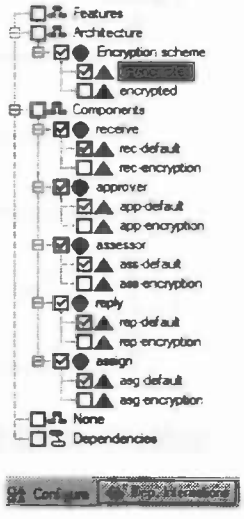

Figure 5.11.: Configuring a product (process) in COVAMOF. By selecting a variant on the architecture level, those on the components level are automatically selected.

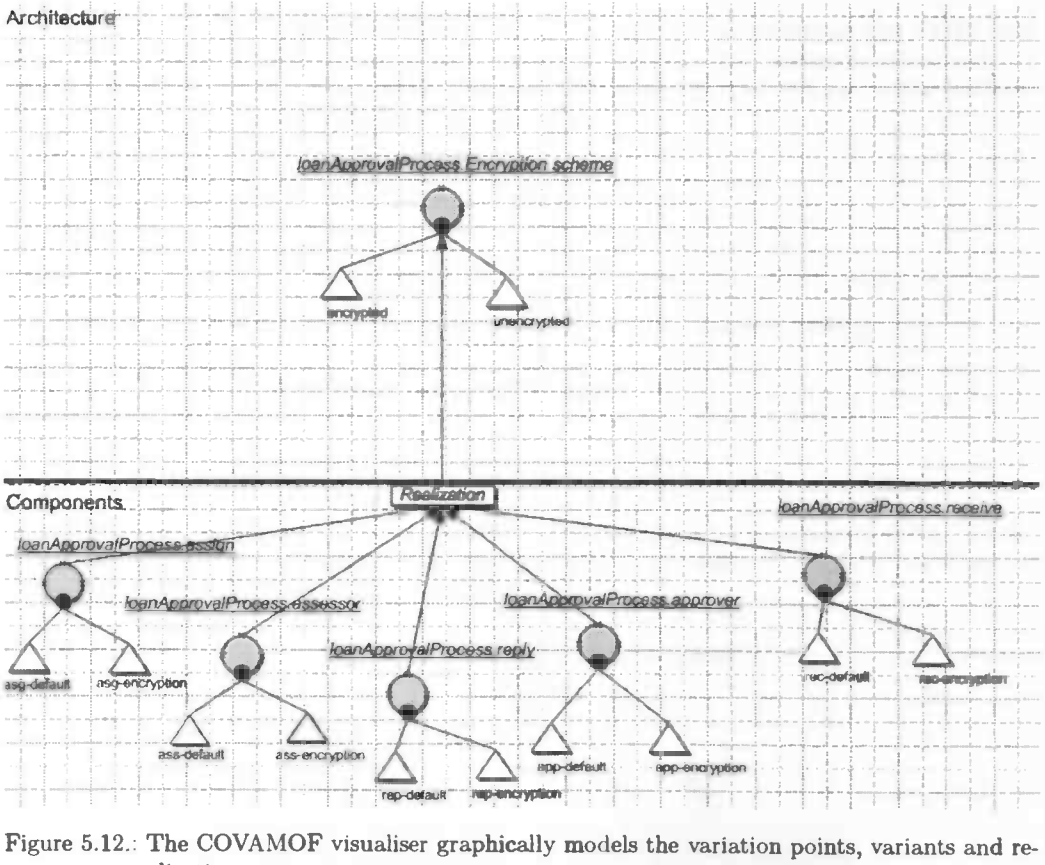

Figure 5.12.: The COVAMOF visualiser graphically models the variation points, variants and realizations.

6. Testing VxBPEL

6.1. Testing procedure

In order to show that VxBPEL combined with the ActiveBPEL modifications and COVAMOF plug-in indeed offers a solution to the questions posed in section 3.4, several testcases will now be used. Each of these testcases answers a specific (part of a) question as given in the testcase's description. Each testcase will consist of the following:

- A description of the testcase and what it will show or prove.
- A graphical representation of the testcase if needed.
- The expected behaviour in order to show that the testcase was successful.
- The observed behaviour when running the testcase.
- A conclusion for this testcase.

The graphical representation will use the symbols as used in figure 6.1 to indicate a variation point and its variants. These are approximately the same symbols as used in COVAMOF, as there is of yet no specific graphical notation for VxBPEL.

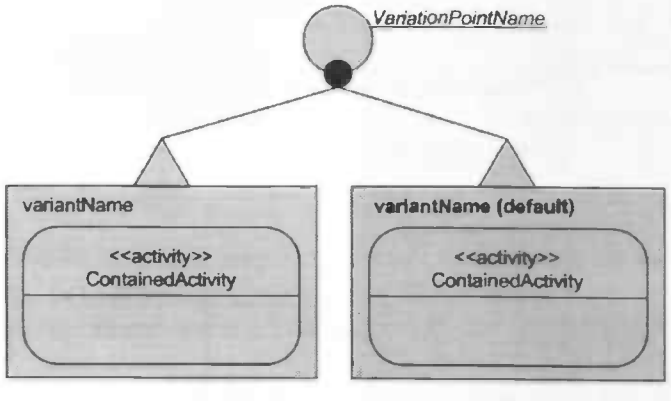

Figure 6.1.: Notation of variation points and variants in process definitions

In this notation, a variation point is denoted by a circle. The name of the variation point is shown underlined and italic. Attached to these variation points are variants (denoted by triangles) with a rectangle showing the activities contained in this variant. The name is placed at the topleft corner in the rectangle, with the default variant's name in bold.

6.2. Testcases

6.2.1. Deploying a VxBPEL process

Description

The first few testcases, up to and including 6.2.4, are given to show that it is possible to deploy, configure and execute a variable process in the VxBPEL engine implementation.

This testcase will show that a VxBPEL process is able to deploy successfully and that the default configuration for each CVP (configurable variation point) is indeed automatically configured after deployment.

Graphical representation

The process definition can be graphically modeled as in figure 6.2.

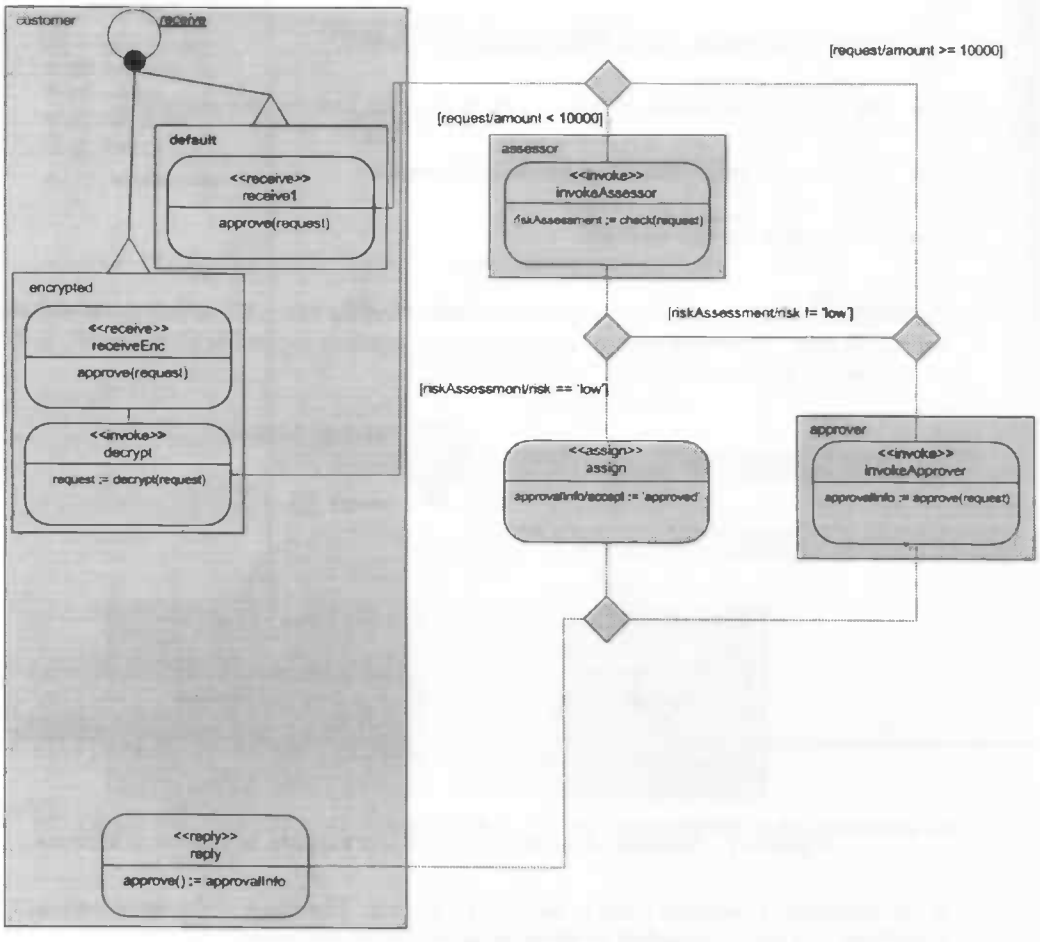

Figure 6.2.: The loan approval process with an encryption variant.

Expected behaviour

After deploying a process to the ActiveBPEL engine, it should be possible to view the process' structure and it should have the process structure that is configured by default.

Observed behaviour

After deployment, the deployed process structure is shown in the ActiveBPEL Administration tool as can be seen in figure 6.3.

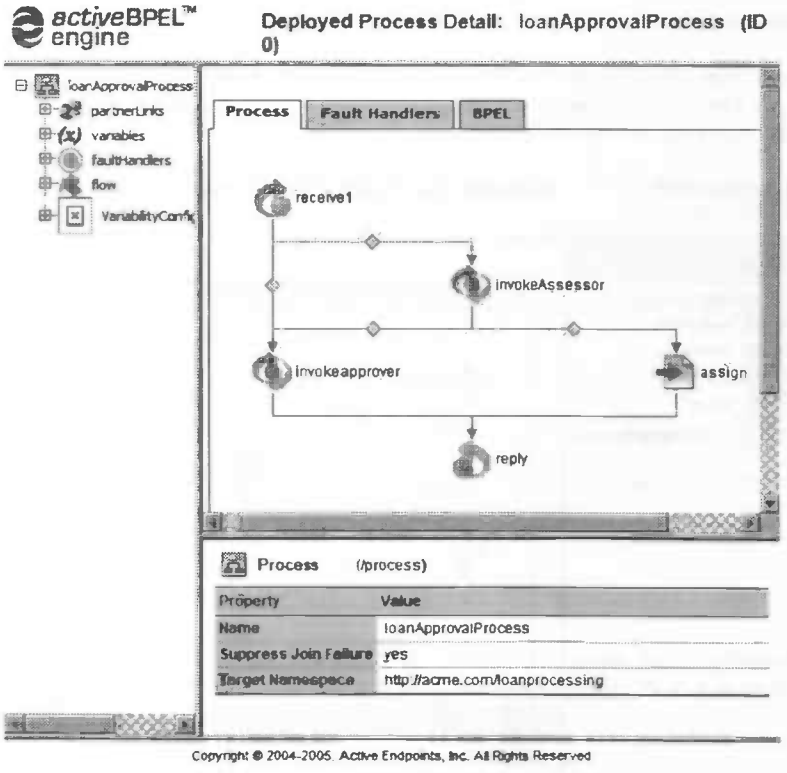

Figure 6.3.: The deployed process' structure.

Conclusion

The structure of the deployed process indeed conforms to the expected structure. Therefore, it is now possible to conclude we can deploy a VxBPEL process.

6.2.2. Running a deployed VxBPEL process

Description

This testcase will show that a VxBPEL process, when just deployed, will be able to run successfully. The VxBPEL process used here will be the same one as used in section 5.4.1: a loan approval process with a variant that receives an encrypted message.

Graphical representation

See section 6.2.1.

Expected behaviour

When deployed, the ActiveBPEL Administration tool should show the process' structure to be the default variant. After execution, the process should have exited normally and the ActiveBPEL Administration tool should show that each of the activities in the default variant were executed.

Observed behaviour

The process exited normally. What the ActiveBPEL Administration tool shows is shown in figure 6.4. As can be seen there, each of the activities for this configuration was executed (as indicated by the checkmarks next to the activities).

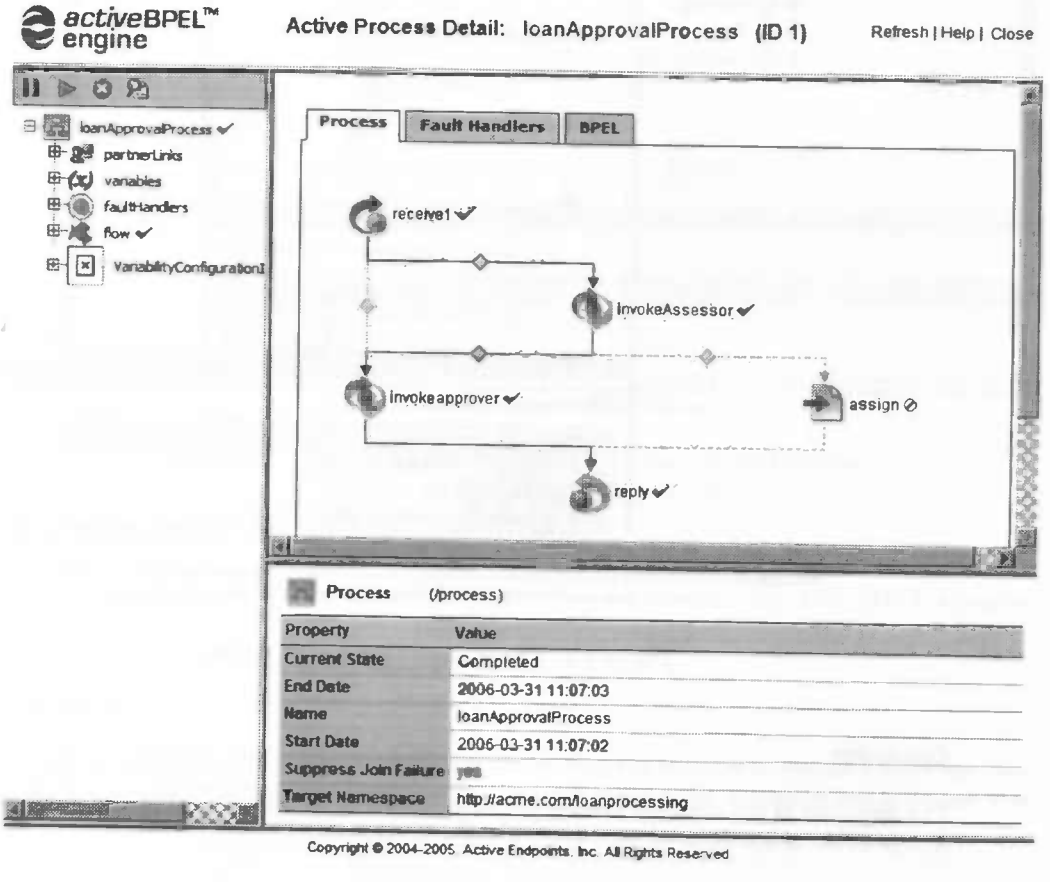

Figure 6.4.: The executed process as shown in the ActiveBPEL Administration tool.

Conclusion

Since the process exited normally and each of the activities for this configuration were executed, it is possible to conclude that it is indeed possible to execute a process immediately after deployment

and that it also executes the correct activities.

6.2.3. Changing a process' configuration

Description

After deployment, it should be possible to manage a process' variability. This means that it should be possible to change the configuration of a process. In the VxBPEL implementation, that is possible through the JMX interface that each CVP exposes.

Graphical representation

See section 6.2.1.

Expected behaviour

After process deployment, a certain structure can be seen in the ActiveBPEL Administration tool. Changing the configuration of a CVP, by using the MX4J adaptor as shown in section 5.4.1 should alter the structure shown for the deployed process.

Observed behaviour

After altering the configured variant name, like in figures 5.4 and 5.5, the ActiveBPEL Administration tool shows the process structure as can be seen in figure 6.5.

Conclusion

After altering the configuration using the JMX interface, the process has a different structure when viewing it in the ActiveBPEL Administration tool, which means that it is indeed possible to change a process' structure.

6.2.4. Executing a different variant

Description

After altering the configuration, any processes started should use the new configuration.

Graphical representation

See section 6.2.1.

Expected behaviour

An executed process should exit normally and the ActiveBPEL Administration tool should show a process was executed with a different structure.

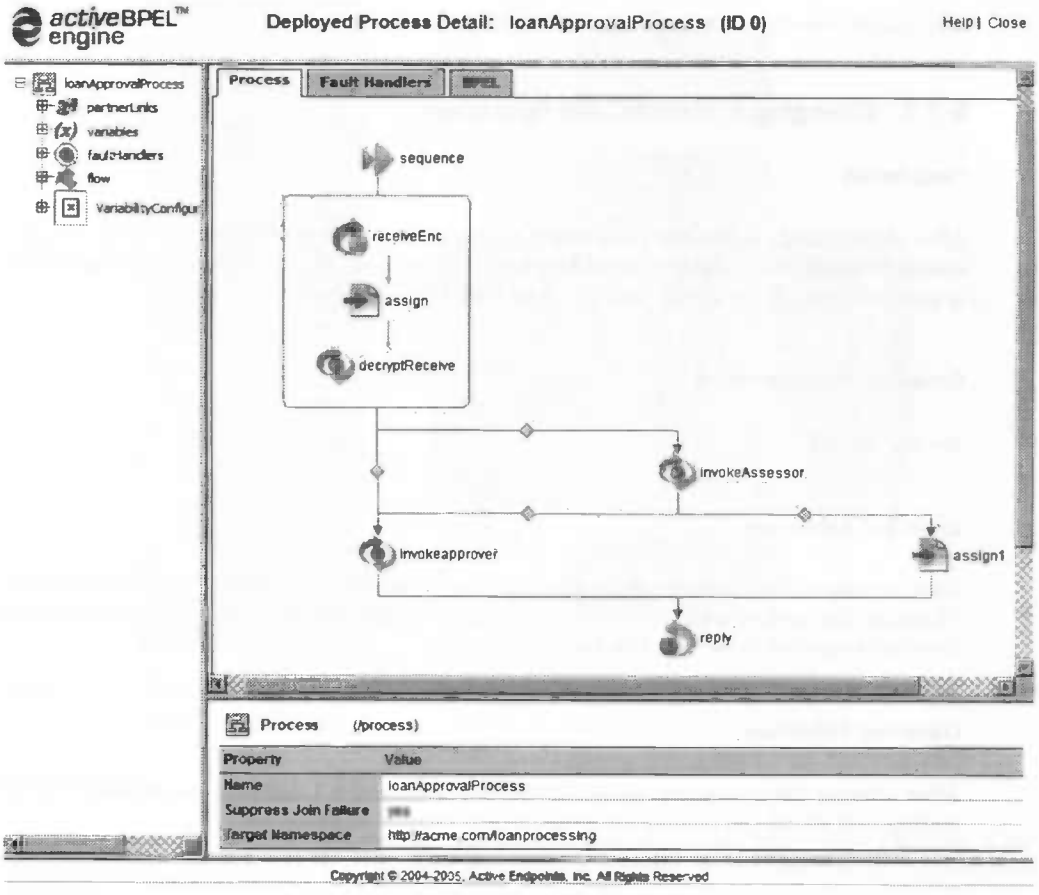

Figure 6.5.: The process structure after changing the configuration.

Observed behaviour

The process exited normally, and figure 6.6 shows what can be seen in the ActiveBPEL Administration tool.

Conclusion

As the ActiveBPEL Administration tool now shows that the encryption variant's activities were executed, we can conclude that after altering the configuration the corresponding activities were executed.

6.2.5. Process consistency (1)

Description

As mentioned in section 3.3 and in research question 1 in section 3.4, process consistency is important. Section 5.1.3 explained why processes should remain consistent during execution, but some process consistency tests will be done to assure this claim.

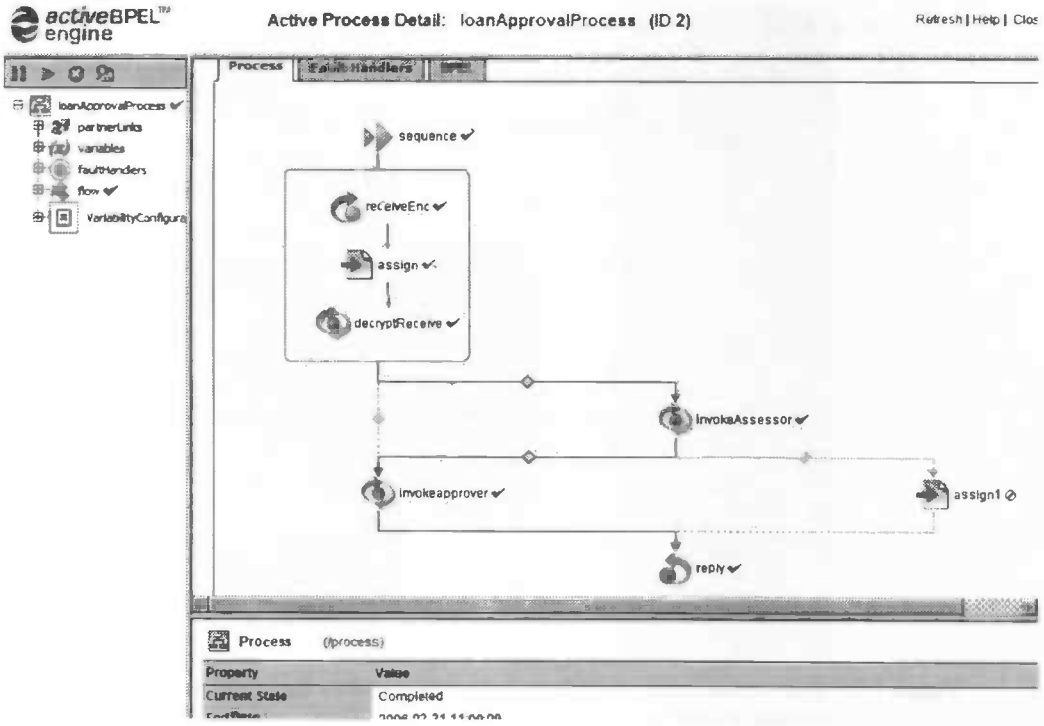

Figure 6.6.: The executed process as shown in the ActiveBPEL Administration tool shows after the process' configuration is changed.

This first example tests whether a currently executing activity is changed if the configuration is altered. In order to do so, for this test a process will be deployed that has an infinite loop in the default configuration. The variant of this process changes the loop contents in such a way that the process will break out of the loop immediately.

The VxBPEL definition can be found in appendix C.2.

Graphical representation

Figure 6.7 depicts a graphical representation of this process.

Expected behaviour

In order to prove that process consistency is maintained in this case, one would expect that a process started in one configuration never exits. In other words, if the process is deployed and a client connects, it should never get an answer (except for a time-out) even if the configuration is altered while waiting for an answer.

Observed behaviour

After starting the client application, there is indeed no answer until a time-out occurs. The ActiveBPEL Administration tool shows what is depicted in figure 6.8. It can be seen that the process has not stopped after reconfiguring it.

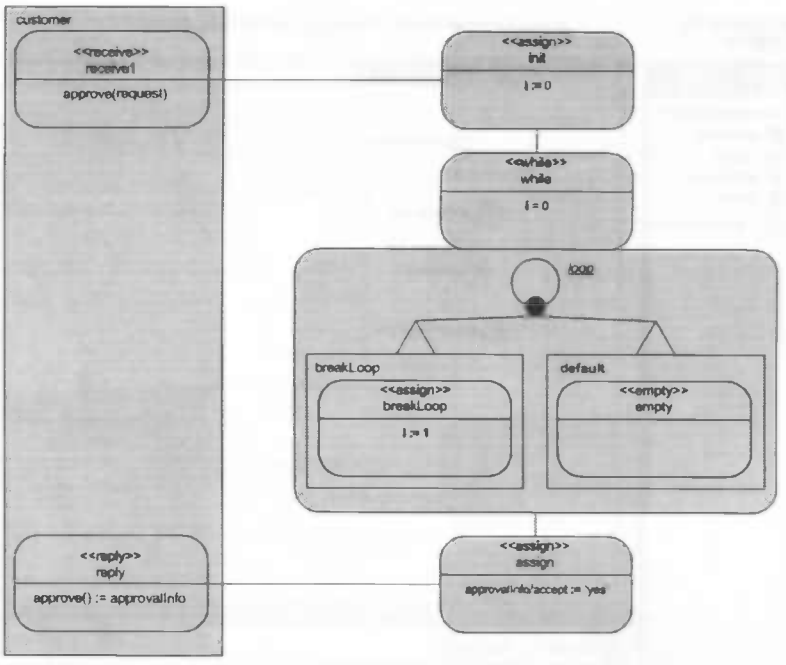

Figure 6.7.: The process runs an infinite loop by default. This loop breaks immediately if the variant is selected.

Conclusion

As the process did not give an answer and the Administration tool shows that the loop is still executing, we can conclude that currently running activities are not affected by configuration changes.

6.2.6. Process consistency (2)

Description

This testcase is similar to the previous. However, in this testcase the process will execute a loop which takes a significant amount of time and as such provides enough time to change the configuration while the loop is executing. After the loop, an answer is returned which depends on the current configuration. This will prove that future activities are also unaffected by configuration changes.

The VxBPEL definition can be found in appendix C.3.

Graphical representation

Figure 6.9 depicts a graphical representation of this process.

Expected behaviour

If process consistency is maintained, when a process is started in a certain configuration, the answer returned after the process finishes should correspond to this configuration. The Administration tool should show this configuration's assign statement was executed.

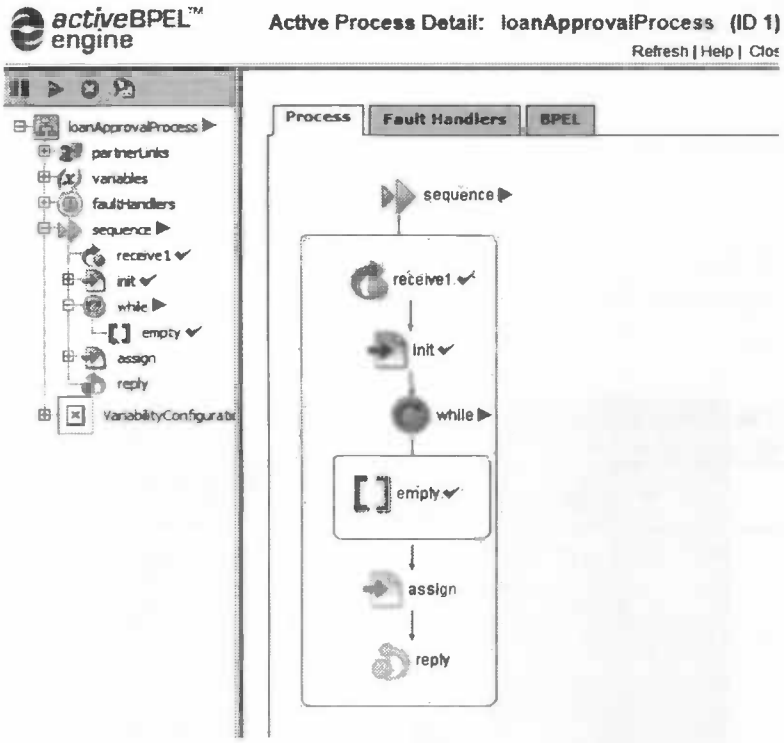

Figure 6.8.: The ActiveBPEL Administration tool shows that the process remains stuck in the infinite loop.

Observed behaviour

The client application is started when the configuration is set to its default and is indeed given the answer which corresponded to this configuration. What the Administration tool shows can be seen in figure 6.10: the default assign statement was executed.

Conclusion

As the answer the process gives corresponds to the configuration in which the process was when it was started and the Administration tool shows the corresponding activities to be executed, we can conclude that future activities are unaffected by configuration changes.

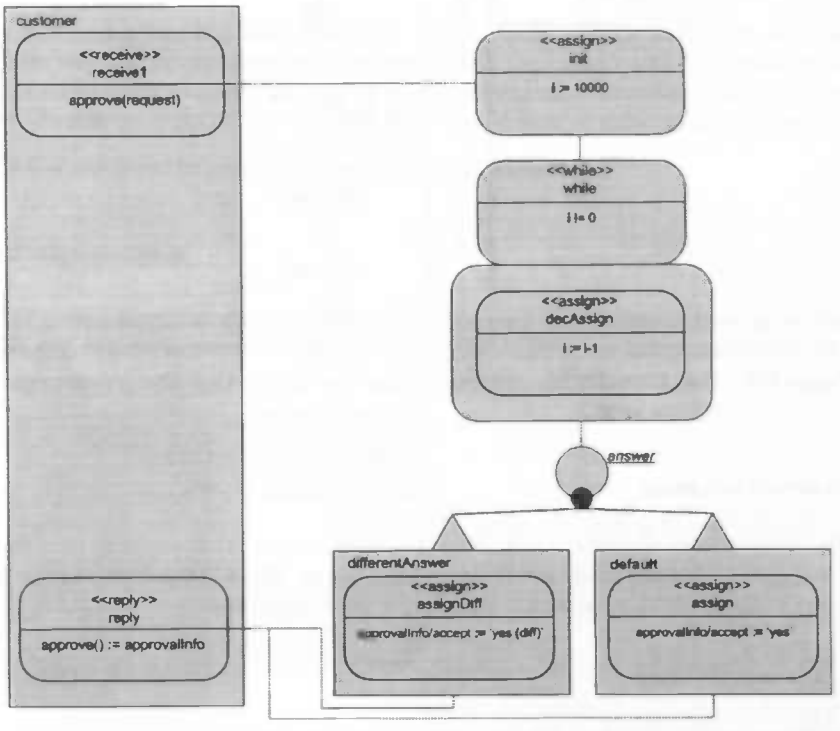

Figure 6.9.: The assign statement after the long loop is different in both variants.

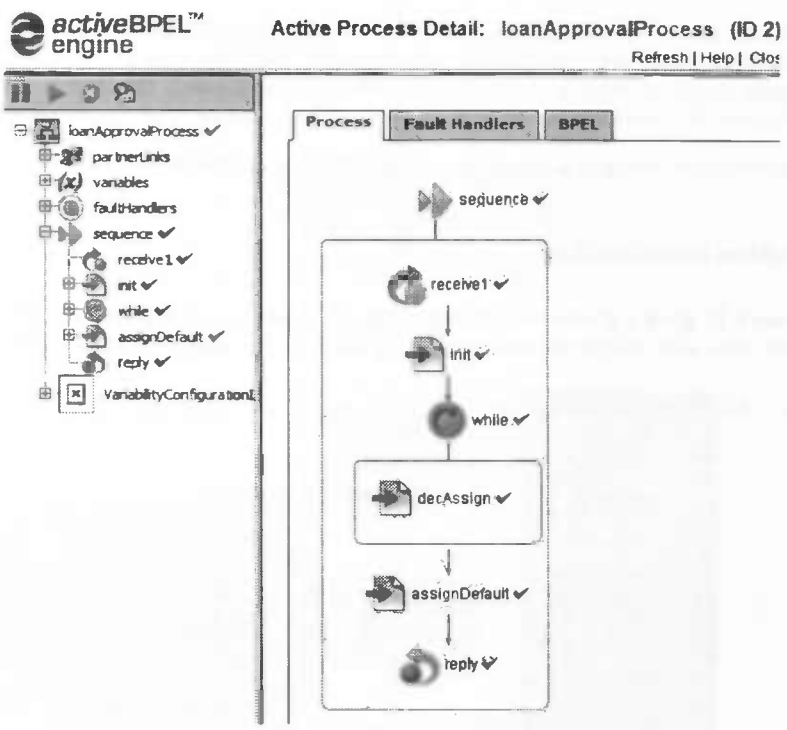

Figure 6.10.: The Administration tool shows that the default assign statement was executed.

6.2.7. BPEL structures

Description

The previous test cases have shown that variation points contained by the `<flow>`, `<sequence>` and `<while>` structures execute successfully. Section 5.1.3 described why it is possible to execute the variation points in containers for multiple activities (e.g., `<flow>`, `<sequence>`) and containers for single activities (e.g., `<while>`, `<case>` and `<otherwise>`). This test case demonstrates the variation point to work for two structures different from the ones used in the previous test cases, being the `<case>` and `<otherwise>` structures.

As every other container activity inherits from one of the interfaces used by the ones tested, there is no need to test the others.

This test case uses a simple process definition that contains a single `<switch>`, with one `<case>` substructure and one `<otherwise>` substructure. Both the `<case>` and the `<otherwise>` contain a single variation point with two variants. Every path through this switch out of the four possible will make the process reply with a different answer, and each of these paths will be tested.

Appendix C.4 contains the corresponding VxBPEL definition.

Graphical representation

Figure 6.11 gives a graphical representation of the test case. Due to space constraints, the contents of the case and otherwise structures is depicted to the right of the associated structure.

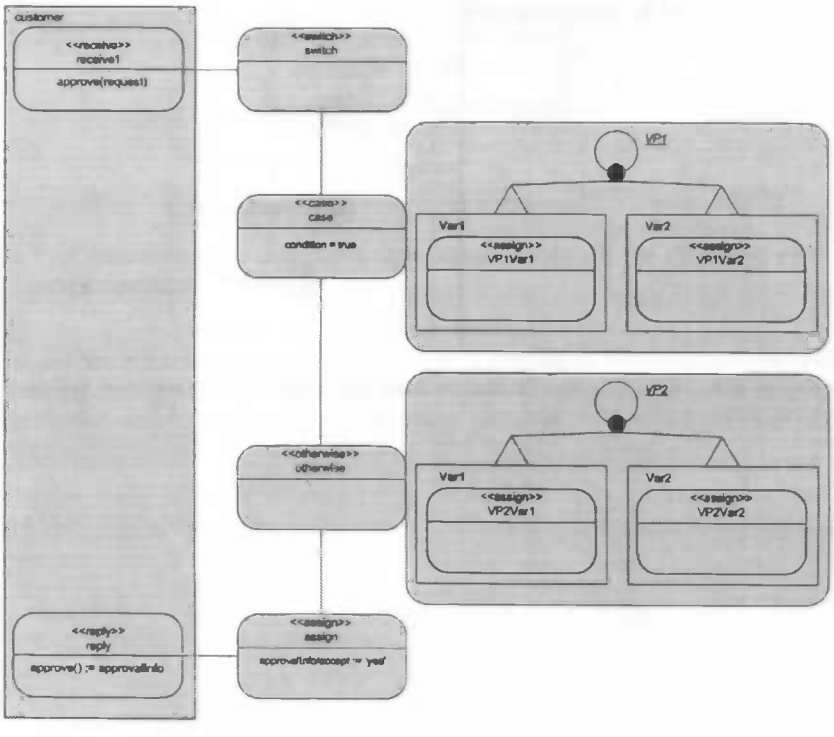

Figure 6.11.: The case and otherwise test case.

Expected behaviour

If the variation point can be executed in both the `<case>` and the `<otherwise>` structure, the process will need to execute correctly each of the four times, with differing answers after each invocation.

Observed behaviour

Every invocation executed correctly and each path produced the correct answer. What the Administration tool shows after each is shown in figures 6.12, 6.13, 6.14 and 6.15.

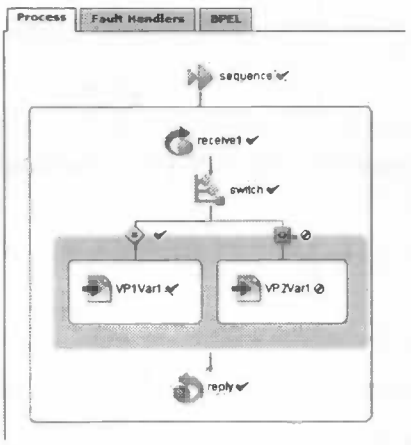

Figure 6.12.: The Administration tool shows that first variant in the case path was correctly executed.

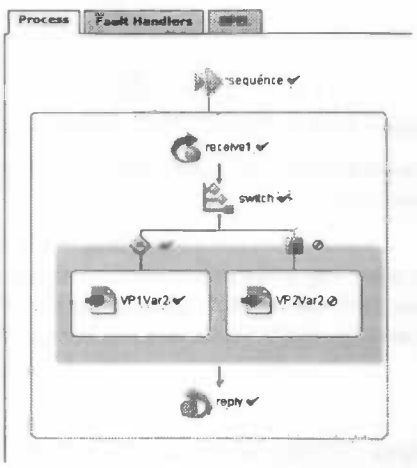

Figure 6.13.: The Administration tool shows that second variant in the case path was correctly executed.

Conclusion

Each invocation of the process executes successfully and each time a different answer is given. The Administration tool shows that each of the four paths was indeed taken. Therefore, taking

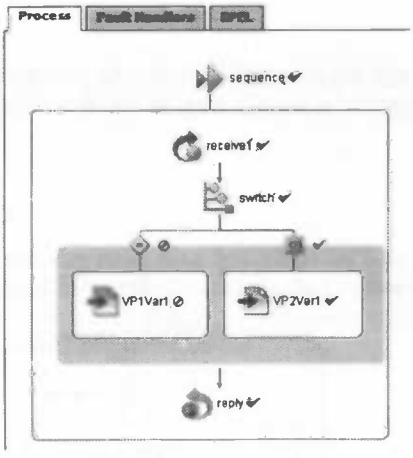

Figure 6.14.: The Administration tool shows that first variant in the otherwise path was correctly executed.

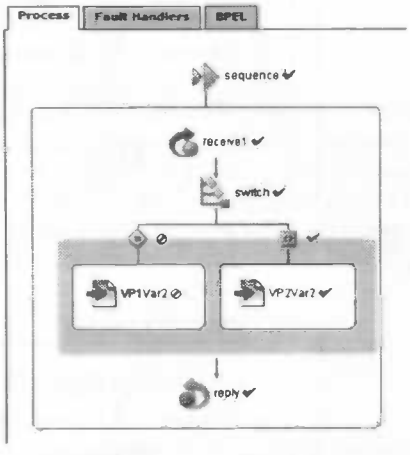

Figure 6.15.: The Administration tool shows that second variant in the otherwise path was correctly executed.

this test case and the other test cases into account, it is possible to conclude that variation points can be successfully executed when contained by `<case>`, `<otherwise>`, `<while>`, `<sequence>` and `<flow>` in particular, and by single activity containers and multiple activity containers in general.

7. Conclusion

7.1. Conclusion

Now that the VxBPEL implementation and the testcases have been presented, we return to the research questions as presented in section 3.4. For easy reference, the questions will be reiterated here.

The central question of this thesis was "How can run-time variability and variability management be allowed, modeled and supported in a service-centric system, so it can quickly react or be adapted to changes in its dynamic environment?", supported by the following research questions:

- ▷ Is it possible to make a service-centric system variable at run-time? How can processes be kept consistent internally?
- ▷ How can (run-time) variability be supported in a service centric system?
- ▷ How can the variability types as described in section 3.2 be captured in this modeling?
- ▷ How can abstraction (i.e., realization relation information) be supported?
- ▷ How can the variability information be modeled graphically in order to give a clear view on the variability in the system?
- ▷ Using modeling of variability information, how can a service-centric system's variability be made manageable externally and at run-time?

The following sections will each discuss one of these research questions, and then the central question will be answered.

Run-time variability and consistency

As shown in chapter 6, the VxBPEL language and the modification to the ActiveBPEL engine allow a variable process to be deployed, configured at run-time and executed while keeping the processes internally consistent.

Modeling (run-time) variability in a service centric system

As demonstrated in chapter 4, VxBPEL provides a way to model variability in a process definition.

Modeling variability types

Section 4.5 demonstrated for each of the variability types in section 3.2 how these could be modeled, without being restricted to just these types.

Realization relations

In chapter 4, it was discussed how a `RequiredConfiguration` could be added to every variant of a high-level variation point, defining which `VPChoices` should be made in order to allow this high level variant to be executed.

Graphical modeling

Section 5.5 demonstrated how variability information can be modeled graphically with COVAMOF when a VxBPEL definition is read in. This gives a better and easier to understand overview of variability than the original VxBPEL code.

Variability management

How management of a deployed VxBPEL is possible with the current implementation has been shown in section 5.3. This is possible externally, through a web browser, at run-time.

Central question

This thesis demonstrated that with VxBPEL it is possible to model variability information in a process, to execute several configurations of processes extended with variability information and to allow the configuration of a deployed process to be managed at run-time and from an external source (web browser). Therefore we can conclude that VxBPEL offers a solution to the central question "How can run-time variability and variability management be allowed, modeled and supported in a service-centric system, so it can quickly react or be adapted to changes in its dynamic environment?".

7.2. Reflection and future work

Even though VxBPEL offers a basic solution to supporting variability in web services, there are several things that I would have liked to do or that should be done to improve it.

First of all, several of the issues that occurred during the implementation phase have made me reconsider that the variability extensions should not be tightly integrated into ActiveBPEL. For example, the validation process as it is now will work for VxBPEL due to several work-arounds. Even though I have left as many validations in place where possible, I think it would be best to completely change the validation process to be able to also validate the VxBPEL processes for things different from "normal" BPEL validation should these be needed.

As another example of an issue related to the non-tight integration with ActiveBPEL engine that occurred after implementation, regarded the ActiveBPEL Administration webtool. The problem with this tool is that it assumes the set of activities in the process definition and the execution are the same. This is true for normal BPEL, but when a process' definition can change, different instances can have different process structures, and this assumption is no longer correct. This does not prevent one from using this tool to see the process structure, but the activities in the execution model are still mapped to their process definition counterparts. This means that when this tool is used after execution of a process and after altering the process' configuration (and as such its definition) to view the process that was executed before the reconfiguration, the tool will display the new definition, with certain activities being untouched. An example can be seen in figure 7.1. This issue did not arise for anything else, but it would be better if an executed process

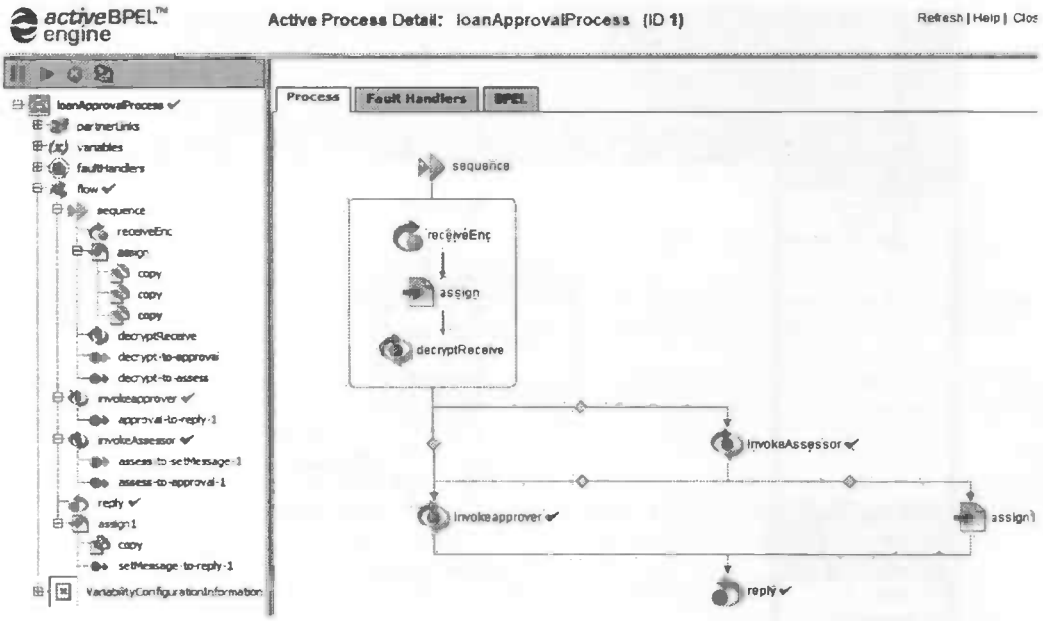

Figure 7.1.: The view of an executed process after the configuration has been changed. The process has completed successfully, but several activities are not checked off, indicating they were never executed.

would store the definition used when the process was instantiated to solve this type of problem. This would also mean adapting the webtool for this change.

Secondly, the implementation as of now has no robust exception and error handling during parsing and reading in process definitions, meaning that an incorrect VxBPEL definition can give several unclear (to one unfamiliar with VxBPEL) problems.

Another implementation issue not yet resolved is that the manageable objects are not removed when a process is removed from the engine. This makes it currently impossible to redeploy a process when changes have been made to the process.

It would also be desirable to develop graphical tooling to design VxBPEL processes, such as the graphic tool ActiveBPEL has for developing normal BPEL processes called ActiveBPEL Designer. However, this tool is not open source and I was therefore unable to change it for this project. It would be extremely useful to design a VxBPEL process in a way similar to what is shown in figure 7.2, which is an edited screenshot of ActiveBPEL Designer.

Another thing I would like to change is the exact terminology used by the VxBPEL implementation. As COVAMOF was added to this project in a late stage, several concepts are very similar (VPChoices in VxBPEL and Bindings for RealizationRelation rules in COVAMOF) but do not use the same name, which can be confusing.

Related to the previous remark is the absence of integration with COVAMOF. As COVAMOF has very useful tooling available, it would be desirable to be able to reconfigure VxBPEL processes from COVAMOF tooling instead of from a tool completely external from ActiveBPEL and COVAMOF.

Also, VxBPEL currently has a rather basic modeling of variation points. For example, it is not possible (without work-arounds) to implement a pure optional variation point, a variation point that is present in one configuration but absent in another. At the moment, this needs to be modeled in VxBPEL by an `<empty>` statement for the case where the variation point is absent, needlessly complicating the process definition.

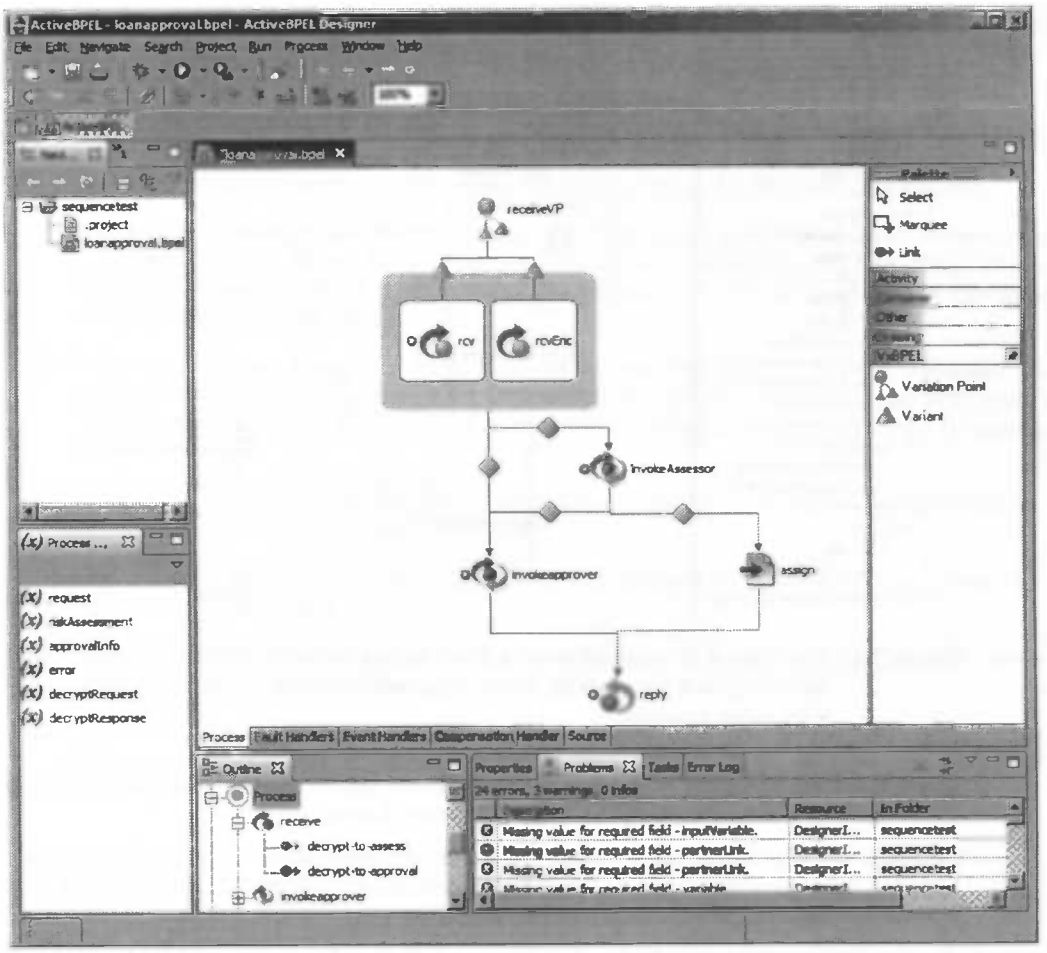

Figure 7.2.: An edited screenshot of ActiveBPEL designer, showing how graphically designing VxBPEL processes could be done.

Open variation points (variation points to which variants can be added at run-time) is also an addition to VxBPEL that would be extremely valuable. It is technically possible to do this through JMX, though most likely not via the MX4J Http/Adaptor tool used for defining variability currently. There is also the issue of introducing new services to be invoked and of introducing new variables to the system, among others.

Finally, 3.3 discussed how run-time variability for the purposes of this thesis was defined as variability between process invocations, while not affecting the structure of a running process. It may be desirable, however, to be able to alter the structure of a running process indeed, which requires a more complicated and detailed modeling of variability. It will also require a redesign of ActiveBPEL's modification, as during execution variability is no longer visible with the current design.

Conclusively, VxBPEL is useful as a proof of concept but much still needs to be done in order to make it viable for industrial applications.

Bibliography

[1] F. Bachmann and L. Bass. Managing variability in software architectures. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'01)*, 2001.

[2] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems With Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), October 1992.

[3] BEA, IBM, Microsoft, SAP AG, and Siebel Systems. Business Process Execution Language for Web Services V1.1 specification. Available from several of the partners' web pages, e.g. `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`, May 2003.

[4] Jan Bosch. *Design & Use of Software Architectures.* Addison-Wesley, 2000.

[5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Description Language (WSDL) 1.1. Published on the web at `http://www.w3.org/TR/wsdl`, March 2001.

[6] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[7] Active Endpoints. ActiveBPEL engine. Website: `http://www.activebpel.org`.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[9] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success.* Addison-Wesley, 1997.

[10] OASIS. UDDI Version 3 Specification. The latest version of this specification can be found at `http://uddi.org/pubs/uddi_v3.htm`. Several other UDDI-related specifications can be accessed through `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`, October 2004.

[11] OASIS. Web Services Business Process Execution Language Version 2.0 Committee Draft. The latest version of this draft is downloadable through `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`, March 2006.

[12] OMG. Unified Modeling Language (UML), version 2.0. The specification can be accessed via `http://www.omg.org/technology/documents/formal/uml.htm`, August 2005.

[13] Oracle. Order booking tutorial. PDF, `http://www.oracle.com/technology/products/ias/bpel/htdocs/1012_support.html#docs`, direct download link `http://download.oracle.com/otndocs/products/bpel/orderbooking.pdf`, May 2005.

[14] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.

[15] Johanneke Siljee, Ivor Bosloper, Jos Nijhuis, and Dieter Hammer. DySOA: making Service Systems Self-Adaptive. In *International Conference on Service-Oriented Computing (ICSOC'05)*, Amsterdam, the Netherlands, 2005.

[16] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In *The Third Software Product Line Conference (SPLC 2004)*, Boston, USA, 2004.

[17] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Modeling Dependencies in Product Families with COVAMOF. In *Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2006)*, March 2006.

[18] N. Yasemin Topaloglu and Rafael Capilla. Modeling the Variability of Web Services from a Pattern Point of View. In *Web Services, European Conference, ECOWS 2004, Erfurt, Germany, September 27-30, 2004, Proceedings*, volume 3250 of *Lecture Notes in Computer Science*, pages 128–138, 2004.

[19] W.T. Tsai, Weiwei Song, Ray Paul, Zhibin Cao, and Hai Huang. Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing. In *28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, Hongkong, 2004.

[20] W3C. XML Path Language (XPath) Version 1.0. Published on the web at `http://www.w3.org/TR/xpath`, November 1999.

[21] W3C. Simple Object Access Protocol (SOAP). Published on the web at `http://www.w3.org/TR/soap`, June 2003.

[22] W3C. Extensible Markup Language (XML) 1.0 (Third Edition). Published on the web at `http://www.w3.org/TR/REC-xml`, February 2004.

[23] W3C. Web services architecture. Working Group Note, url: `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`, February 2004.

Appendix

A. Definition of terms, acronyms and abbreviations

This chapter gives a quick overview of terms, acronyms and abbreviations, sorted alphabetically, that are not explicitly explained throughout this thesis.

- HTTP: Hypertext Transfer Protocol, the set of rules for exchanging files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web.
- OASIS: Organization for the Advancement of Structured Information Standards, a global consortium that drives the development of e-business and web service standards.
- SOAP: Simple Object Access Protocol. SOAP is a lightweight XML based protocol used for invoking web services and exchanging structured data and type information on the Web[21].
- UDDI: Universal Description Discovery and Integration, a directory model for web services. UDDI is a specification for maintaining standardized directories of information about web services, recording their capabilities, location and requirements in a universally recognized format[10].
- UML: Unified Modeling Language, a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system. It is standardized by the Object Management Group, a computing industry collaboration to promote object-oriented interoperability among heterogeneous computing environments. Using UML, developers and architects can make a blueprint of a software project.
- W3C: World Wide Web Consortium, the international governing body for web standards.
- WSDL: Web Services Description Language, an XML-formatted language used to describe a Web service's capabilities as collections of communication endpoints capable of exchanging messages. WSDL is an integral part of UDDI, an XML-based worldwide business registry. WSDL is the language that UDDI uses. WSDL was developed jointly by Microsoft and IBM[5].
- XML: Extensible Markup Language. A flexible way to create common information formats and share both the format and the data on the World Wide Web, intranets, and elsewhere. XML is a formal recommendation from the W3C[22].
- XPath: XML Path Language, a language that describes how to locate specific elements (and attributes, processing instructions, etc.) in an XML document[20].

A. Definition of terms, acronyms and abbreviations

B. VxBPEL design

B.1. Original BPEL file

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
BPEL Process Definition
Edited using ActiveWebflow(tm) Designer version 1.0.0 (http://www.active-endpoints.com)
-->
<process name="loanApprovalProcess" suppressJoinFailure="yes" targetNamespace="http://acme.com/loanprocessing"
        xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:apns="http://tempuri.org/services/loanapprover"
        xmlns:asns="http://tempuri.org/services/loanassessor"
        xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:lns="http://loans.org/wsdl/loan-approval"
        xmlns:loandef="http://tempuri.org/services/loandefinitions"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <partnerLinks>
      <partnerLink myRole="approver" name="customer" partnerLinkType="lns:loanApprovalLinkType"/>
      <partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
      <partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor"/>
   </partnerLinks>
   <variables>
      <variable messageType="loandef:creditInformationMessage" name="request"/>
      <variable messageType="asns:riskAssessmentMessage" name="riskAssessment"/>
      <variable messageType="apns:approvalMessage" name="approvalInfo"/>
      <variable messageType="loandef:loanRequestErrorMessage" name="error"/>
   </variables>
   <faultHandlers>
      <catch faultName="apns:loanProcessFault" faultVariable="error">
         <reply faultName="apns:loanProcessFault" operation="approve" partnerLink="customer"
                portType="apns:loanApprovalPT" variable="error"/>
      </catch>
   </faultHandlers>
   <flow>
      <links>
         <link name="receive-to-approval"/>
         <link name="receive-to-assess"/>
         <link name="approval-to-reply"/>
         <link name="assess-to-setMessage"/>
         <link name="assess-to-approval"/>
         <link name="setMessage-to-reply"/>
      </links>
      <receive createInstance="yes" name="receive1" operation="approve" partnerLink="customer"
               portType="apns:loanApprovalPT" variable="request">
         <source linkName="receive-to-approval"
                 transitionCondition="bpws:getVariableData('request', 'amount')&gt;=10000"/>
         <source linkName="receive-to-assess"
                 transitionCondition="bpws:getVariableData('request', 'amount')&lt;10000"/>
      </receive>
      <invoke inputVariable="request" name="invokeapprover" operation="approve" outputVariable="approvalInfo"
              partnerLink="approver" portType="apns:loanApprovalPT">
         <target linkName="receive-to-approval"/>
         <target linkName="assess-to-approval"/>
         <source linkName="approval-to-reply"/>
      </invoke>
      <invoke inputVariable="request" name="invokeAssessor" operation="check" outputVariable="riskAssessment"
              partnerLink="assessor" portType="asns:riskAssessmentPT">
         <target linkName="receive-to-assess"/>
         <source linkName="assess-to-setMessage"
                 transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
         <source linkName="assess-to-approval"
                 transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
      </invoke>
      <reply name="reply" operation="approve" partnerLink="customer" portType="apns:loanApprovalPT"
             variable="approvalInfo">
         <target linkName="approval-to-reply"/>
         <target linkName="setMessage-to-reply"/>
      </reply>
```

```
      <assign name="assign">
         <target linkName="assess-to-setMessage"/>
         <source linkName="setMessage-to-reply"/>
         <copy>
            <from expression="'approved'"/>
            <to part="accept" variable="approvalInfo"/>
         </copy>
      </assign>
   </flow>
</process>
```

B.2. Separate variability information

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
BPEL Process Definition
Edited using ActiveWebflow(tm) Designer version 1.0.0 (http://www.active-endpoints.com)
-->
<process name="loanApprovalProcess" suppressJoinFailure="yes" targetNamespace="http://acme.com/loanprocessing"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:apns="http://tempuri.org/services/loanapprover"
         xmlns:asns="http://tempuri.org/services/loanassessor"
         xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:lns="http://loans.org/wsdl/loan-approval"
         xmlns:loandef="http://tempuri.org/services/loandefinitions"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:vxbpel="http://vxbpel.rug.org">
   <partnerLinks>
      <partnerLink myRole="approver" name="customer" partnerLinkType="lns:loanApprovalLinkType"/>
      <partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
      <partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor"/>
   </partnerLinks>
   <variables>
      <variable messageType="loandef:creditInformationMessage" name="request"/>
      <variable messageType="asns:riskAssessmentMessage" name="riskAssessment"/>
      <variable messageType="apns:approvalMessage" name="approvalInfo"/>
      <variable messageType="loandef:loanRequestErrorMessage" name="error"/>
   </variables>
   <faultHandlers>
      <catch faultName="apns:loanProcessFault" faultVariable="error">
         <reply faultName="apns:loanProcessFault" operation="approve" partnerLink="customer"
                portType="apns:loanApprovalPT" variable="error"/>
      </catch>
   </faultHandlers>
   <flow>
      <links>
         <link name="receive-to-approval"/>
         <link name="receive-to-assess"/>
         <link name="approval-to-reply"/>
         <link name="assess-to-setMessage"/>
         <link name="assess-to-approval"/>
         <link name="setMessage-to-reply"/>
      </links>
      <receive createInstance="yes" name="receive1" operation="approve" partnerLink="customer"
               portType="apns:loanApprovalPT" variable="request">
         <source linkName="receive-to-approval"
                 transitionCondition="bpws:getVariableData('request', 'amount')&gt;=10000"/>
         <source linkName="receive-to-assess"
                 transitionCondition="bpws:getVariableData('request', 'amount')&lt;10000"/>
      </receive>
      <invoke inputVariable="request" name="invokeapprover" operation="approve" outputVariable="approvalInfo"
              partnerLink="approver" portType="apns:loanApprovalPT">
         <target linkName="receive-to-approval"/>
         <target linkName="assess-to-approval"/>
         <source linkName="approval-to-reply"/>
      </invoke>
      <invoke inputVariable="request" name="invokeAssessor" operation="check" outputVariable="riskAssessment"
              partnerLink="assessor" portType="asns:riskAssessmentPT">
         <target linkName="receive-to-assess"/>
         <source linkName="assess-to-setMessage"
                 transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
         <source linkName="assess-to-approval"
                 transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
      </invoke>
      <reply name="reply" operation="approve" partnerLink="customer" portType="apns:loanApprovalPT"
             variable="approvalInfo">
```

```
      <target linkName="approval-to-reply"/>
      <target linkName="setMessage-to-reply"/>
   </reply>
   <assign name="assign">
      <target linkName="assess-to-setMessage"/>
      <source linkName="setMessage-to-reply"/>
      <copy>
         <from expression="'approved'"/>
         <to part="accept" variable="approvalInfo"/>
      </copy>
   </assign>
</flow>

<vxbpel:VariabilityConfigurationInformation>
   <vxbpel:VariationPoints>
      <vxbpel:VariationPoint name="VP1" path="/process/flow/invoke[@name='invokeAssessor']">
         <vxbpel:Variants>
            <vxbpel:Variant name="default">
               <vxbpel:VPBpelCode>
                  <invoke inputVariable="request" name="invokeAssessor" operation="check"
                          outputVariable="riskAssessment" partnerLink="assessor"
                          portType="asns:riskAssessmentPT">
                     <target linkName="receive-to-assess"/>
                     <source linkName="assess-to-setMessage"
                             transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
                     <source linkName="assess-to-approval"
                             transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
                  </invoke>
               </vxbpel:VPBpelCode>
            </vxbpel:Variant>
            <vxbpel:Variant name="alternative1">
               <vxbpel:VPBpelCode>
                  <sequence name="sequence">
                     <target linkName="receive-to-sequence"/>
                     <source linkName="sequence-to-setMessage"
                             transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
                     <source linkName="sequence-to-approval"
                             transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
                     <!-- Code here to copy contents of original request message to approvalRequest variable.
                     -->
                     <assign name="copy1">
                        <copy>
                           <from expression="'Request for approval'"/>
                           <to part="notes" variable="approvalRequest"/>
                        </copy>
                     </assign>
                     <invoke inputVariable="approvalRequest" name="invokeAssessor" operation="check"
                             outputVariable="riskAssessment" partnerLink="assessor"
                             portType="asns:riskAssessmentPT">
                     </invoke>
                  </sequence>
               </vxbpel:VPBpelCode>
            </vxbpel:Variant>
         </vxbpel:Variants>
      </vxbpel:VariationPoint>
   </vxbpel:VariationPoints>
   <vxbpel:ConfigurableVariationPoints>
      <vxbpel:ConfigurableVariationPoint id="CVP1" defaultVariant="default">
         <vxbpel:Rationale>
            <!-- Here information should be put that explains the goal of this VP
                 and what consequences configuring this VP has.
                 This information could be formal or informal. Formal information
                 would allow autonomous or automatic configuration. -->
         </vxbpel:Rationale>
         <vxbpel:Variants>
            <vxbpel:Variant name="default">
               <!-- The name of the variant is now identical to the variant names of the
                    associated VPs. This is not assumed to always be the case. -->
               <vxbpel:VariantInfo>
                  <!--  Information is put here that pertains only to this variant. -->
               </vxbpel:VariantInfo>
               <vxbpel:RequiredConfiguration>
                  <vxbpel:VPChoices>
                     <vxbpel:VPChoice vpname="links" variant="default"/>
                     <vxbpel:VPChoice vpname="to-invoke-approval" variant="default"/>
                     <vxbpel:VPChoice vpname="VP1" variant="default"/>
                     <vxbpel:VPChoice vpname="to-setMessage" variant="default"/>
                  </vxbpel:VPChoices>
               </vxbpel:RequiredConfiguration>
            </vxbpel:Variant>
```

```
            <vxbpel:Variant name="alternative1">
               <vxbpel:VariantInfo>
                  <!-- empty for now. -->
               </vxbpel:VariantInfo>
               <vxbpel:RequiredConfiguration>
                  <vxbpel:VPChoices>
                     <vxbpel:VPChoice vpname="variables" variant="alternative1"/>
                     <vxbpel:VPChoice vpname="links" variant="alternative1"/>
                     <vxbpel:VPChoice vpname="to-invoke-approval" variant="alternative1"/>
                     <vxbpel:VPChoice vpname="VP1" variant="alternative1"/>
                     <vxbpel:VPChoice vpname="to-setMessage" variant="alternative1"/>
                  </vxbpel:VPChoices>
               </vxbpel:RequiredConfiguration>
            </vxbpel:Variant>
         </vxbpel:Variants>
      </vxbpel:ConfigurableVariationPoint>
   </vxbpel:ConfigurableVariationPoints>
</vxbpel:VariabilityConfigurationInformation>

</process>
```

B.3. Inline variability information

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
BPEL Process Definition
Edited using ActiveWebflow(tm) Designer version 1.0.0 (http://www.active-endpoints.com)
-->

<!-- This file has added variability information needed for the SeCSE project. It was added inline. -->

<process name="loanApprovalProcess" suppressJoinFailure="yes" targetNamespace="http://acme.com/loanprocessing"
        xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:apns="http://tempuri.org/services/loanapprover"
        xmlns:asns="http://tempuri.org/services/loanassessor"
        xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:lns="http://loans.org/wsdl/loan-approval"
        xmlns:loandef="http://tempuri.org/services/loandefinitions"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:vxbpel="http://vxbpel.rug.org">
   <partnerLinks>
      <partnerLink myRole="approver" name="customer" partnerLinkType="lns:loanApprovalLinkType"/>
      <partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
      <partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor"/>
   </partnerLinks>
   <variables>
      <variable messageType="loandef:creditInformationMessageAlternative" name="approvalRequest"/>
      <variable messageType="loandef:creditInformationMessage" name="request"/>
      <variable messageType="asns:riskAssessmentMessage" name="riskAssessment"/>
      <variable messageType="apns:approvalMessage" name="approvalInfo"/>
      <variable messageType="loandef:loanRequestErrorMessage" name="error"/>
   </variables>
   <faultHandlers>
      <catch faultName="apns:loanProcessFault" faultVariable="error">
         <reply faultName="apns:loanProcessFault" operation="approve" partnerLink="customer"
                portType="apns:loanApprovalPT" variable="error"/>
      </catch>
   </faultHandlers>
   <flow>
      <links>
         <link name="receive-to-approval"/>
         <link name="receive-to-assess"/>
         <link name="assess-to-setMessage"/>
         <link name="assess-to-approval"/>
         <link name="receive-to-sequence"/>
         <link name="sequence-to-setMessage"/>
         <link name="sequence-to-approval"/>
         <link name="approval-to-reply"/>
         <link name="setMessage-to-reply"/>
      </links>
      <receive createInstance="yes" name="receive1" operation="approve" partnerLink="customer"
               portType="apns:loanApprovalPT" variable="request">
         <source linkName="receive-to-approval"
                 transitionCondition="bpws:getVariableData('request', 'amount')&gt;=10000"/>
         <source linkName="receive-to-assess"
                 transitionCondition="bpws:getVariableData('request', 'amount')&lt;10000"/>
```

```
   </receive>
   <invoke inputVariable="request" name="invokeapprover" operation="approve" outputVariable="approvalInfo"
           partnerLink="approver" portType="apns:loanApprovalPT">
      <target linkName="receive-to-approval"/>
      <target linkName="assess-to-approval"/>
      <target linkName="sequence-to-approval"/>
      <source linkName="approval-to-reply"/>
   </invoke>
   <!-- The current way of notation would mean breaking the BPEL format - as unsupported tags are ignored
        along with their children (Section 6.3 in the WS-BPEL Draft). However, an XSLT transformation
        should yield valid BPEL again. -->
   <vxbpel:VariationPoint name="VP1">
      <vxbpel:Variants>
         <vxbpel:Variant name="default">
            <vxbpel:VPBpelCode>
               <invoke inputVariable="request" name="invokeAssessor" operation="check"
                       outputVariable="riskAssessment" partnerLink="assessor"
                       portType="asns:riskAssessmentPT">
                  <target linkName="receive-to-assess"/>
                  <source linkName="assess-to-setMessage"
                          transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
                  <source linkName="assess-to-approval"
                          transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
               </invoke>
            </vxbpel:VPBpelCode>
         </vxbpel:Variant>
         <vxbpel:Variant name="alternative1">
            <vxbpel:VPBpelCode>
               <sequence name="sequence">
                  <target linkName="receive-to-sequence"/>
                  <source linkName="sequence-to-setMessage"
                          transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
                  <source linkName="sequence-to-approval"
                          transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
                  <!-- Code here to copy contents of original request message to approvalRequest variable.
                  -->
                  <assign name="copy1">
                     <copy>
                        <from expression="'Request for approval'"/>
                        <to part="notes" variable="approvalRequest"/>
                     </copy>
                  </assign>
                  <invoke inputVariable="approvalRequest" name="invokeAssessor" operation="check"
                          outputVariable="riskAssessment" partnerLink="assessor"
                          portType="asns:riskAssessmentPT">
                  </invoke>
               </sequence>
            </vxbpel:VPBpelCode>
         </vxbpel:Variant>
      </vxbpel:Variants>
   </vxbpel:VariationPoint>
   <reply name="reply" operation="approve" partnerLink="customer" portType="apns:loanApprovalPT"
          variable="approvalInfo">
      <target linkName="approval-to-reply"/>
      <target linkName="setMessage-to-reply"/>
   </reply>
   <assign name="assign">
      <target linkName="assess-to-setMessage"/>
      <target linkName="sequence-to-setMessage"/>
      <source linkName="setMessage-to-reply"/>
      <copy>
         <from expression="'approved'"/>
         <to part="accept" variable="approvalInfo"/>
      </copy>
   </assign>
</flow>

<vxbpel:VariabilityConfigurationInformation>
   <vxbpel:ConfigurableVariationPoints>
      <vxbpel:ConfigurableVariationPoint id="CVP1" defaultVariant="default">
         <vxbpel:Rationale>
            <!-- Here information should be put that explains the goal of this VP
                 and what consequences configuring this VP has.
                 This information could be formal or informal. Formal information
                 would allow autonomous or automatic configuration. -->
         </vxbpel:Rationale>
         <vxbpel:Variants>
            <vxbpel:Variant name="default">
               <!-- The name of the variant is now identical to the variant names of the
                    associated VPs. This is not assumed to always be the case. -->
```

```
                <vxbpel:VariantInfo>
                   <!--  Information is put here that pertains only to this variant. -->
                </vxbpel:VariantInfo>
                <vxbpel:RequiredConfiguration>
                   <vxbpel:VPChoices>
                      <vxbpel:VPChoice vpname="VP1" variant="default"/>
                   </vxbpel:VPChoices>
                </vxbpel:RequiredConfiguration>
             </vxbpel:Variant>
             <vxbpel:Variant name="alternative1">
                <vxbpel:VariantInfo>
                   <!-- empty for now. -->
                </vxbpel:VariantInfo>
                <vxbpel:RequiredConfiguration>
                   <vxbpel:VPChoices>
                      <vxbpel:VPChoice vpname="VP1" variant="alternative1"/>
                   </vxbpel:VPChoices>
                </vxbpel:RequiredConfiguration>
             </vxbpel:Variant>
          </vxbpel:Variants>
       </vxbpel:ConfigurableVariationPoint>
    </vxbpel:ConfigurableVariationPoints>
  </vxbpel:VariabilityConfigurationInformation>
</process>
```

C. Samples code

C.1. Loan approval example

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Variable BPEL Process Definition
Edited from a sample for ActiveBPEL
-->
<process name="loanApprovalProcess" suppressJoinFailure="yes" targetNamespace="http://acme.com/loanprocessing"
       xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
       xmlns:apns="http://tempuri.org/services/loanapprover"
       xmlns:asns="http://tempuri.org/services/loanassessor"
       xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
       xmlns:lns="http://loans.org/wsdl/loan-approval"
       xmlns:loandef="http://tempuri.org/services/loandefinitions"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:vxbpel="http://vxbpel.rug.org"
       xmlns:ns1="http://encryptionPLT"
       xmlns:enc="http://tempuri.org/services/encryptionService">
   <partnerLinks>
      <partnerLink myRole="approver" name="customer" partnerLinkType="lns:loanApprovalLinkType"/>
      <partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
      <partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor"/>
      <partnerLink name="encryptionServiceLinkType" partnerLinkType="ns1:encryptionServiceLinkType"
                   partnerRole="encryption"/>
   </partnerLinks>
   <variables>
      <variable messageType="loandef:creditInformationMessage" name="request"/>
      <variable messageType="asns:riskAssessmentMessage" name="riskAssessment"/>
      <variable messageType="apns:approvalMessage" name="approvalInfo"/>
      <variable messageType="apns:approvalMessage" name="encryptedApprovalInfo"/>
      <variable messageType="loandef:loanRequestErrorMessage" name="error"/>
      <variable messageType="enc:decryptRequest" name="decryptRequest"/>
      <variable messageType="enc:decryptResponse" name="decryptResponse"/>
   </variables>
   <faultHandlers>
      <catch faultName="apns:loanProcessFault" faultVariable="error">
         <reply faultName="apns:loanProcessFault" operation="approve" partnerLink="customer"
                portType="apns:loanApprovalPT" variable="error"/>
      </catch>
   </faultHandlers>
   <flow>
      <links>
         <link name="receive-to-approval"/>
         <link name="receive-to-assess"/>
         <link name="decrypt-to-approval"/>
         <link name="decrypt-to-assess"/>
         <link name="approval-to-reply"/>
         <link name="approval-to-reply-1"/>
         <link name="assess-to-setMessage"/>
         <link name="assess-to-setMessage-1"/>
         <link name="assess-to-approval"/>
         <link name="assess-to-approval-1"/>
         <link name="setMessage-to-reply"/>
         <link name="setMessage-to-reply-1"/>
      </links>
      <vxbpel:VariationPoint name="receive">
         <vxbpel:Variants>
            <vxbpel:Variant name="default">
               <receive createInstance="yes" name="receive1" operation="approve" partnerLink="customer"
                        portType="apns:loanApprovalPT" variable="request">
                  <source linkName="receive-to-approval"
                     transitionCondition="bpws:getVariableData('request', 'amount')&gt;=10000"/>
                  <source linkName="receive-to-assess"
                     transitionCondition="bpws:getVariableData('request', 'amount')&lt;10000"/>
               </receive>
            </vxbpel:Variant>
            <vxbpel:Variant name="encryption">
               <sequence>
```

```
            <receive createInstance="yes" name="receiveEnc" operation="approve" partnerLink="customer"
                  portType="apns:loanApprovalPT" variable="request">
            </receive>
            <assign>
               <copy>
                  <from part="firstName" variable="request"/>
                  <to part="firstName" variable="decryptRequest"/>
               </copy>
               <copy>
                  <from part="name" variable="request"/>
                  <to part="name" variable="decryptRequest"/>
               </copy>
               <copy>
                  <from part="amount" variable="request"/>
                  <to part="amount" variable="decryptRequest"/>
               </copy>
            </assign>
            <invoke inputVariable="decryptRequest" name="decryptReceive" operation="decrypt"
                    outputVariable="decryptResponse" partnerLink="encryptionServiceLinkType"
                    portType="enc:EncryptionWebService">
            </invoke>
            <source linkName="decrypt-to-approval"
                    transitionCondition="bpws:getVariableData('decryptResponse',
                    'creditInformationMessage', 'creditInformationMessage/amount')&gt;=10000"/>
            <source linkName="decrypt-to-assess"
                    transitionCondition="bpws:getVariableData('decryptResponse',
                    'creditInformationMessage', 'creditInformationMessage/amount')&lt;10000"/>
         </sequence>
      </vxbpel:Variant>
   </vxbpel:Variants>
</vxbpel:VariationPoint>
<vxbpel:VariationPoint name="approver">
   <vxbpel:Variants>
      <vxbpel:Variant name="default">
         <invoke inputVariable="request" name="invokeapprover" operation="approve"
                 outputVariable="approvalInfo" partnerLink="approver" portType="apns:loanApprovalPT">
            <target linkName="receive-to-approval"/>
            <target linkName="assess-to-approval"/>
            <source linkName="approval-to-reply"/>
         </invoke>
      </vxbpel:Variant>
      <vxbpel:Variant name="encryption">
         <invoke inputVariable="request" name="invokeapprover" operation="approve"
                 outputVariable="approvalInfo" partnerLink="approver" portType="apns:loanApprovalPT">
            <target linkName="decrypt-to-approval"/>
            <target linkName="assess-to-approval-1"/>
            <source linkName="approval-to-reply-1"/>
         </invoke>
      </vxbpel:Variant>
   </vxbpel:Variants>
</vxbpel:VariationPoint>
<vxbpel:VariationPoint name="assessor">
   <vxbpel:Variants>
      <vxbpel:Variant name="default">
         <invoke inputVariable="request" name="invokeAssessor" operation="check"
                 outputVariable="riskAssessment" partnerLink="assessor" portType="asns:riskAssessmentPT">
            <target linkName="receive-to-assess"/>
            <source linkName="assess-to-setMessage"
                 transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
            <source linkName="assess-to-approval"
                 transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
         </invoke>
      </vxbpel:Variant>
      <vxbpel:Variant name="encryption">
          <invoke inputVariable="request" name="invokeAssessor" operation="check"
                  outputVariable="riskAssessment" partnerLink="assessor" portType="asns:riskAssessmentPT">
            <target linkName="decrypt-to-assess"/>
            <source linkName="assess-to-setMessage-1"
                 transitionCondition="bpws:getVariableData('riskAssessment', 'risk')='low'"/>
            <source linkName="assess-to-approval-1"
                 transitionCondition="bpws:getVariableData('riskAssessment', 'risk')!='low'"/>
         </invoke>
      </vxbpel:Variant>
   </vxbpel:Variants>
</vxbpel:VariationPoint>
<vxbpel:VariationPoint name="reply">
   <vxbpel:Variants>
      <vxbpel:Variant name="default">
         <reply name="reply" operation="approve" partnerLink="customer" portType="apns:loanApprovalPT"
                variable="approvalInfo">
```

```
            <target linkName="approval-to-reply"/>
            <target linkName="setMessage-to-reply"/>
          </reply>
        </vxbpel:Variant>
        <vxbpel:Variant name="encryption">
          <reply name="reply" operation="approve" partnerLink="customer" portType="apns:loanApprovalPT"
                 variable="approvalInfo">
            <target linkName="approval-to-reply-1"/>
            <target linkName="setMessage-to-reply-1"/>
          </reply>
        </vxbpel:Variant>
      </vxbpel:Variants>
    </vxbpel:VariationPoint>
    <vxbpel:VariationPoint name="assign">
      <vxbpel:Variants>
        <vxbpel:Variant name="default">
          <assign name="assign">
            <target linkName="assess-to-setMessage"/>
            <source linkName="setMessage-to-reply"/>
            <copy>
              <from expression="'approved'"/>
              <to part="accept" variable="approvalInfo"/>
            </copy>
          </assign>
        </vxbpel:Variant>
        <vxbpel:Variant name="encryption">
          <assign name="assign1">
            <target linkName="assess-to-setMessage-1"/>
            <source linkName="setMessage-to-reply-1"/>
            <copy>
              <from expression="'approved'"/>
              <to part="accept" variable="approvalInfo"/>
            </copy>
          </assign>
        </vxbpel:Variant>
      </vxbpel:Variants>
    </vxbpel:VariationPoint>
</flow>

<vxbpel:VariabilityConfigurationInformation>
    <vxbpel:ConfigurableVariationPoints>
      <vxbpel:ConfigurableVariationPoint id="encryption" defaultVariant="unencrypted">
        <vxbpel:Name>
          Encryption scheme
        </vxbpel:Name>
        <vxbpel:Rationale>
          It is possible to configure the loan approval process to support encryption.
        </vxbpel:Rationale>
        <vxbpel:Variants>
          <vxbpel:Variant name="unencrypted">
            <vxbpel:VariantInfo>
              Information is received unencrypted.
            </vxbpel:VariantInfo>
            <vxbpel:RequiredConfiguration>
              <vxbpel:VPChoices>
                <vxbpel:VPChoice vpname="receive" variant="default"/>
                <vxbpel:VPChoice vpname="approver" variant="default"/>
                <vxbpel:VPChoice vpname="assessor" variant="default"/>
                <vxbpel:VPChoice vpname="reply" variant="default"/>
                <vxbpel:VPChoice vpname="assign" variant="default"/>
              </vxbpel:VPChoices>
            </vxbpel:RequiredConfiguration>
          </vxbpel:Variant>
          <vxbpel:Variant name="encrypted">
            <vxbpel:VariantInfo>
              Information will be decrypted after receipt. The customer's privacy is better protected
              this way.
            </vxbpel:VariantInfo>
            <vxbpel:RequiredConfiguration>
              <vxbpel:VPChoices>
                <vxbpel:VPChoice vpname="receive" variant="encryption"/>
                <vxbpel:VPChoice vpname="approver" variant="encryption"/>
                <vxbpel:VPChoice vpname="assessor" variant="encryption"/>
                <vxbpel:VPChoice vpname="reply" variant="encryption"/>
                <vxbpel:VPChoice vpname="assign" variant="encryption"/>
              </vxbpel:VPChoices>
            </vxbpel:RequiredConfiguration>
          </vxbpel:Variant>
        </vxbpel:Variants>
      </vxbpel:ConfigurableVariationPoint>
```

```
      </vxbpel:ConfigurableVariationPoints>
   </vxbpel:VariabilityConfigurationInformation>
</process>
```

C.2. Process consistency (1)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Variable BPEL Process Definition
Edited from a sample for ActiveBPEL
-->
<process name="loanApprovalProcess" suppressJoinFailure="yes"
         targetNamespace="http://acme.com/loanprocessing"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:apns="http://tempuri.org/services/loanapprover"
         xmlns:asns="http://tempuri.org/services/loanassessor"
         xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:lns="http://loans.org/wsdl/loan-approval"
         xmlns:loandef="http://tempuri.org/services/loandefinitions"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:vxbpel="http://vxbpel.rug.org"
         xmlns:ns1="http://encryptionPLT" xmlns:enc="http://tempuri.org/services/encryptionService">
   <partnerLinks>
      <partnerLink myRole="approver" name="customer" partnerLinkType="lns:loanApprovalLinkType"/>
      <partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
      <partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor"/>
      <partnerLink name="encryptionServiceLinkType" partnerLinkType="ns1:encryptionServiceLinkType"
                   partnerRole="encryption"/>
   </partnerLinks>
   <variables>
      <variable type="xsd:integer" name="i"/>
      <variable messageType="loandef:creditInformationMessage" name="request"/>
      <!--<variable messageType="asns:riskAssessmentMessage" name="riskAssessment"/>-->
      <variable messageType="apns:approvalMessage" name="approvalInfo"/>
      <!--<variable messageType="apns:approvalMessage" name="encryptedApprovalInfo"/>-->
      <variable messageType="loandef:loanRequestErrorMessage" name="error"/>
      <!--<variable messageType="enc:decryptRequest" name="decryptRequest"/>
      <variable messageType="enc:decryptResponse" name="decryptResponse"/>-->
   </variables>
   <faultHandlers>
      <catch faultName="apns:loanProcessFault" faultVariable="error">
         <reply faultName="apns:loanProcessFault" operation="approve" partnerLink="customer"
                portType="apns:loanApprovalPT" variable="error"/>
      </catch>
   </faultHandlers>
   <sequence>
      <receive createInstance="yes" name="receive1" operation="approve" partnerLink="customer"
               portType="apns:loanApprovalPT" variable="request">
      </receive>
      <assign name="init">
         <copy>
            <from expression="0"/>
            <to variable="i"/>
         </copy>
      </assign>
      <while condition="bpws:getVariableData('i') = 0">
         <vxbpel:VariationPoint name="vp1">
            <vxbpel:Variants>
               <vxbpel:Variant name="vp1.var1">
                  <empty name="empty"/>
               </vxbpel:Variant>
               <vxbpel:Variant name="vp1.var2">
                  <assign name="breakLoop">
                     <copy>
                        <from expression="1"/>
                        <to variable="i"/>
                     </copy>
                  </assign>
               </vxbpel:Variant>
            </vxbpel:Variants>
         </vxbpel:VariationPoint>
      </while>
      <assign name="assign">
         <copy>
            <from expression="'yes'"/>
            <to part="accept" variable="approvalInfo"/>
         </copy>
```

```
      </assign>
      <reply name="reply" operation="approve" partnerLink="customer" portType="apns:loanApprovalPT"
            variable="approvalInfo">
      </reply>
   </sequence>

   <vxbpel:VariabilityConfigurationInformation>
      <vxbpel:ConfigurableVariationPoints>
         <vxbpel:ConfigurableVariationPoint id="loop" defaultVariant="loop">
            <vxbpel:Name>Looping</vxbpel:Name>
            <vxbpel:Rationale>This implementation can have an infinite loop for testing whether
                              running processes are affected.</vxbpel:Rationale>
            <vxbpel:Variants>
               <vxbpel:Variant name="loop">
                  <vxbpel:VariantInfo>Process loops for a very long time.</vxbpel:VariantInfo>
                  <vxbpel:RequiredConfiguration>
                     <vxbpel:VPChoices>
                        <vxbpel:VPChoice vpname="vp1" variant="vp1.var1"/>
                     </vxbpel:VPChoices>
                  </vxbpel:RequiredConfiguration>
               </vxbpel:Variant>
               <vxbpel:Variant name="noloop">
                  <vxbpel:VariantInfo>No looping.</vxbpel:VariantInfo>
                  <vxbpel:RequiredConfiguration>
                     <vxbpel:VPChoices>
                        <vxbpel:VPChoice vpname="vp1" variant="vp1.var2"/>
                     </vxbpel:VPChoices>
                  </vxbpel:RequiredConfiguration>
               </vxbpel:Variant>
            </vxbpel:Variants>
         </vxbpel:ConfigurableVariationPoint>
      </vxbpel:ConfigurableVariationPoints>
   </vxbpel:VariabilityConfigurationInformation>
</process>
```

C.3. Process consistency (2)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Variable BPEL Process Definition
Edited from a sample for ActiveBPEL
-->
<process name="loanApprovalProcess" suppressJoinFailure="yes"
         targetNamespace="http://acme.com/loanprocessing"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:apns="http://tempuri.org/services/loanapprover"
         xmlns:asns="http://tempuri.org/services/loanassessor"
         xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:lns="http://loans.org/wsdl/loan-approval"
         xmlns:loandef="http://tempuri.org/services/loandefinitions"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:vxbpel="http://vxbpel.rug.org"
         xmlns:ns1="http://encryptionPLT" xmlns:enc="http://tempuri.org/services/encryptionService">
   <partnerLinks>
      <partnerLink myRole="approver" name="customer" partnerLinkType="lns:loanApprovalLinkType"/>
      <partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
      <partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor"/>
      <partnerLink name="encryptionServiceLinkType" partnerLinkType="ns1:encryptionServiceLinkType"
                   partnerRole="encryption"/>
   </partnerLinks>
   <variables>
      <variable type="xsd:integer" name="i"/>
      <variable messageType="loandef:creditInformationMessage" name="request"/>
      <!--<variable messageType="asns:riskAssessmentMessage" name="riskAssessment"/>-->
      <variable messageType="apns:approvalMessage" name="approvalInfo"/>
      <!--<variable messageType="apns:approvalMessage" name="encryptedApprovalInfo"/>-->
      <variable messageType="loandef:loanRequestErrorMessage" name="error"/>
      <!--<variable messageType="enc:decryptRequest" name="decryptRequest"/>
      <variable messageType="enc:decryptResponse" name="decryptResponse"/>-->
   </variables>
   <faultHandlers>
      <catch faultName="apns:loanProcessFault" faultVariable="error">
         <reply faultName="apns:loanProcessFault" operation="approve" partnerLink="customer"
                portType="apns:loanApprovalPT" variable="error"/>
      </catch>
   </faultHandlers>
   <sequence>
```

```
      <receive createInstance="yes" name="receive1" operation="approve" partnerLink="customer"
               portType="apns:loanApprovalPT" variable="request">
      </receive>
      <assign name="init">
         <copy>
            <from expression="100000"/>
            <to variable="i"/>
         </copy>
      </assign>
      <while condition="bpws:getVariableData('i') != 0">
         <assign name="decAssign">
            <copy>
               <from expression="bpws:getVariableData('i') - 1"/>
               <to variable="i"/>
            </copy>
         </assign>
      </while>
      <vxbpel:VariationPoint name="vp1">
         <vxbpel:Variants>
            <vxbpel:Variant name="vp1.var1">
               <assign name="assignDefault">
                  <copy>
                     <from expression="'yes'"/>
                     <to part="accept" variable="approvalInfo"/>
                  </copy>
               </assign>
            </vxbpel:Variant>
            <vxbpel:Variant name="vp1.var2">
               <assign name="assignDiff">
                  <copy>
                     <from expression="'yes (diff)'"/>
                     <to part="accept" variable="approvalInfo"/>
                  </copy>
               </assign>
            </vxbpel:Variant>
         </vxbpel:Variants>
      </vxbpel:VariationPoint>
      <reply name="reply" operation="approve" partnerLink="customer" portType="apns:loanApprovalPT"
             variable="approvalInfo">
      </reply>
   </sequence>

   <vxbpel:VariabilityConfigurationInformation>
      <vxbpel:ConfigurableVariationPoints>
         <vxbpel:ConfigurableVariationPoint id="assign" defaultVariant="yes">
            <vxbpel:Name>Looping</vxbpel:Name>
            <vxbpel:Rationale>This implementation changes the assignment after a long loop to
                              see if a running process' structure remains the same when the
                              definition is changed.</vxbpel:Rationale>
            <vxbpel:Variants>
               <vxbpel:Variant name="yes">
                  <vxbpel:VariantInfo>'yes' is assigned as return value.</vxbpel:VariantInfo>
                  <vxbpel:RequiredConfiguration>
                     <vxbpel:VPChoices>
                        <vxbpel:VPChoice vpname="vp1" variant="vp1.var1"/>
                     </vxbpel:VPChoices>
                  </vxbpel:RequiredConfiguration>
               </vxbpel:Variant>
               <vxbpel:Variant name="no">
                  <vxbpel:VariantInfo>A different value is returned.</vxbpel:VariantInfo>
                  <vxbpel:RequiredConfiguration>
                     <vxbpel:VPChoices>
                        <vxbpel:VPChoice vpname="vp1" variant="vp1.var2"/>
                     </vxbpel:VPChoices>
                  </vxbpel:RequiredConfiguration>
               </vxbpel:Variant>
            </vxbpel:Variants>
         </vxbpel:ConfigurableVariationPoint>
      </vxbpel:ConfigurableVariationPoints>
   </vxbpel:VariabilityConfigurationInformation>
</process>
```

C.4. BPEL constructs

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
```

```
Variable BPEL Process Definition
Edited from a sample for ActiveBPEL
-->
<process name="loanApprovalProcess" suppressJoinFailure="yes"
        targetNamespace="http://acme.com/loanprocessing"
        xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:apns="http://tempuri.org/services/loanapprover"
        xmlns:asns="http://tempuri.org/services/loanassessor"
        xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:lns="http://loans.org/wsdl/loan-approval"
        xmlns:loandef="http://tempuri.org/services/loandefinitions"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:vxbpel="http://vxbpel.rug.org" xmlns:ns1="http://encryptionPLT"
        xmlns:enc="http://tempuri.org/services/encryptionService">
   <partnerLinks>
      <partnerLink myRole="approver" name="customer" partnerLinkType="lns:loanApprovalLinkType"/>
      <partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
      <partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor"/>
      <partnerLink name="encryptionServiceLinkType" partnerLinkType="ns1:encryptionServiceLinkType"
                   partnerRole="encryption"/>
   </partnerLinks>
   <variables>
      <variable messageType="loandef:creditInformationMessage" name="request"/>
      <variable messageType="asns:riskAssessmentMessage" name="riskAssessment"/>
      <variable messageType="apns:approvalMessage" name="approvalInfo"/>
      <variable messageType="apns:approvalMessage" name="encryptedApprovalInfo"/>
      <variable messageType="loandef:loanRequestErrorMessage" name="error"/>
      <variable messageType="enc:decryptRequest" name="decryptRequest"/>
      <variable messageType="enc:decryptResponse" name="decryptResponse"/>
   </variables>
   <faultHandlers>
      <catch faultName="apns:loanProcessFault" faultVariable="error">
         <reply faultName="apns:loanProcessFault" operation="approve" partnerLink="customer"
                portType="apns:loanApprovalPT" variable="error"/>
      </catch>
   </faultHandlers>
   <sequence>
      <receive createInstance="yes" name="receive1" operation="approve" partnerLink="customer"
               portType="apns:loanApprovalPT" variable="request"/>
         <switch name="switch">

            <case condition="1=0">
               <vxbpel:VariationPoint name="VP1">
                  <vxbpel:Variants>
                     <vxbpel:Variant name="Var1">
                        <assign name="VP1Var1">
                           <copy>
                              <from expression="'CaseVar1'"/>
                              <to part="accept" variable="approvalInfo"/>
                           </copy>
                        </assign>
                     </vxbpel:Variant>
                     <vxbpel:Variant name="Var2">
                        <assign name="VP1Var2">
                           <copy>
                              <from expression="'CaseVar2'"/>
                              <to part="accept" variable="approvalInfo"/>
                           </copy>
                        </assign>
                     </vxbpel:Variant>
                  </vxbpel:Variants>
               </vxbpel:VariationPoint>
            </case>

            <otherwise>
               <vxbpel:VariationPoint name="VP2">
                  <vxbpel:Variants>
                     <vxbpel:Variant name="Var1">
                        <assign name="VP2Var1">
                           <copy>
                              <from expression="'OtherwiseVar1'"/>
                              <to part="accept" variable="approvalInfo"/>
                           </copy>
                        </assign>
                     </vxbpel:Variant>
                     <vxbpel:Variant name="Var2">
                        <assign name="VP2Var2">
                           <copy>
                              <from expression="'OtherwiseVar2'"/>
                              <to part="accept" variable="approvalInfo"/>
```

```
                        </copy>
                      </assign>
                    </vxbpel:Variant>
                  </vxbpel:Variants>
                </vxbpel:VariationPoint>
              </otherwise>

        </switch>
      <reply name="reply" operation="approve" partnerLink="customer" portType="apns:loanApprovalPT"
            variable="approvalInfo"/>
   </sequence>

   <vxbpel:VariabilityConfigurationInformation>
      <vxbpel:ConfigurableVariationPoints>
         <vxbpel:ConfigurableVariationPoint id="casetest" defaultVariant="var1">
            <vxbpel:Name>vp1</vxbpel:Name>
            <vxbpel:Rationale>Testing 'case'</vxbpel:Rationale>
            <vxbpel:Variants>
               <vxbpel:Variant name="var1">
                  <vxbpel:VariantInfo>none</vxbpel:VariantInfo>
                  <vxbpel:RequiredConfiguration>
                     <vxbpel:VPChoices>
                        <vxbpel:VPChoice vpname="VP1" variant="Var1"/>
                        <vxbpel:VPChoice vpname="VP2" variant="Var1"/>
                     </vxbpel:VPChoices>
                  </vxbpel:RequiredConfiguration>
               </vxbpel:Variant>
               <vxbpel:Variant name="var2">
                  <vxbpel:VariantInfo>none either</vxbpel:VariantInfo>
                  <vxbpel:RequiredConfiguration>
                     <vxbpel:VPChoices>
                        <vxbpel:VPChoice vpname="VP1" variant="Var2"/>
                        <vxbpel:VPChoice vpname="VP2" variant="Var2"/>
                     </vxbpel:VPChoices>
                  </vxbpel:RequiredConfiguration>
               </vxbpel:Variant>
            </vxbpel:Variants>
         </vxbpel:ConfigurableVariationPoint>
      </vxbpel:ConfigurableVariationPoints>
   </vxbpel:VariabilityConfigurationInformation>
</process>
```

D. Patterns

D.1. Visitor pattern

This section briefly describes the Visitor pattern[8].

Suppose an application maintains a structure of objects. On each of the elements of this object structure, a new operation needs to be defined in order to add a feature to this application. There are many different elements in this structure, and each of these needs to be adapted to support this new operation.

It is very costly to adapt the objects' definition for each operation that needs to be added, especially if this happens often and the object structure itself does not change.

The Visitor pattern offers a solution for this. One defines a visitor object, that implements a specific operation for each type of object that can possibly be in the object structure. This object is passed to each of the elements in the structure, and when an element receives such a visitor, it "accepts" the visitor by sending a request to this visitor that encodes its class. The element is also included as an argument. The visitor then executes the operation which is associated to that element class - the operation that would normally have to be implemented in the element's class.

For each operation that needs to be added, a visitor can now be defined which contains the functionality that would normally be spread over all the elements.

For more details, see [8].

D.2. Composite pattern

This section briefly describes the Composite pattern[8].

Suppose an application needs to manipulate objects which can either be singular objects or container objects containing several of these singular objects. Each of these objects has similar operations and need to be manipulated in a similar way, but code needs to treat singular objects and container objects differently.

The Composite pattern describes how to define a recursive composition so this distinction need no longer be made. The key to this pattern is an abstract class that represents both singular and container objects and defines operations that are used by singular classes as well as container classes.

Singular objects implement this interface and define the operations that are specific to them. Container objects also implement this interface, but define operations that are specific to them as well as the operations that are specific to singular objects. In the container objects, the operations for singular objects are defined as calling this operation on each of the contained objects.

This allows the application to treat container objects in the same way as singular objects, and also allows container objects to treat their children uniformly, be they singular or container objects.

This pattern is described in more detail in [8].

D. Patterns

E. File and package structure of VxBPEL's ActiveBPEL engine adaptation

E.1. File structure

Figure E.1 contains the file structure of the ActiveBPEL engine source including VxBPEL's modifications.

E.2. Package structure

The package structure can be derived from the file structure in figure E.1, but figure E.2 shows the package structure of the VxBPEL additions by itself.

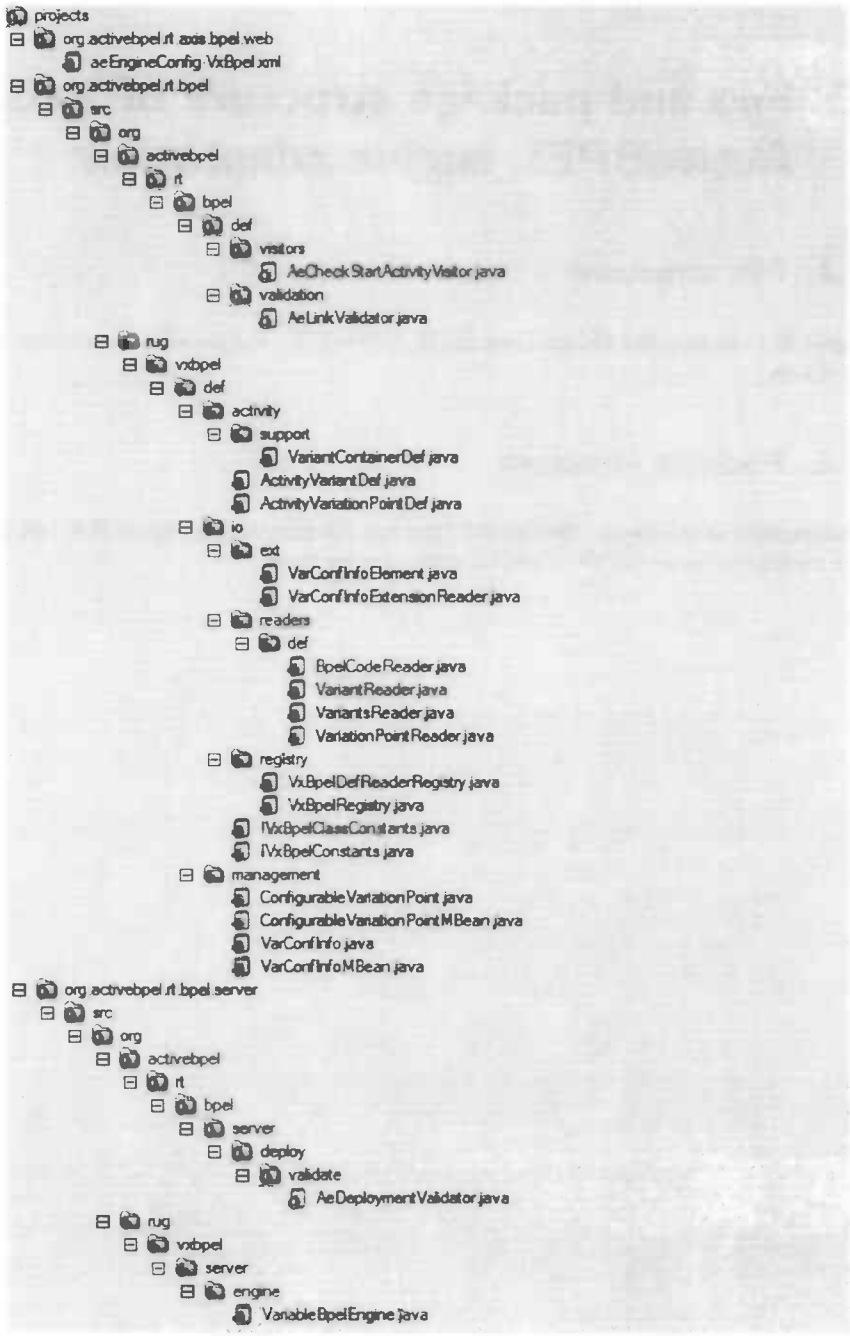

Figure E.1.: The file structure of the ActiveBPEL project with VxBPEL modifications. New files and folders are depicted by a plus sign, modified files by an exclamation mark surrounded by a circle. Folders and files that are unmodified are left out of this view.

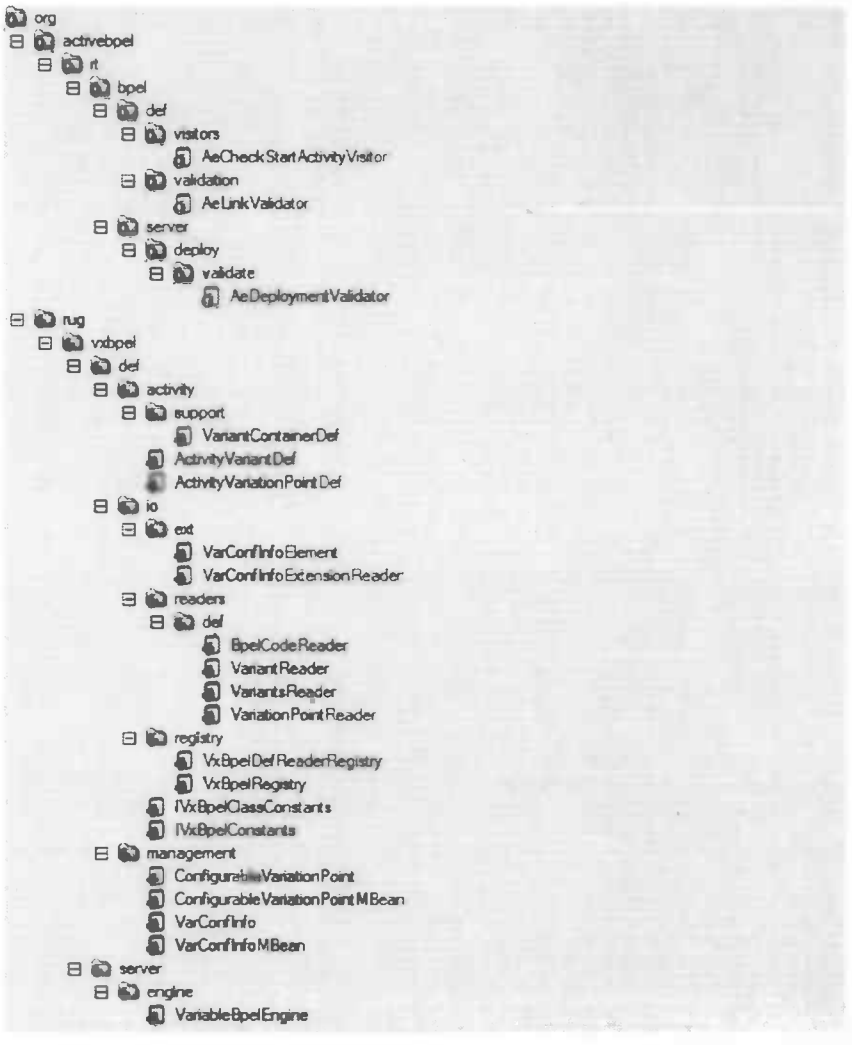

Figure E.2.: The package structure of the ActiveBPEL project with VxBPEL modifications. New classes and packages are depicted by a plus sign, modified classes by an exclamation mark surrounded by a circle. Packages and classes that are unmodified are left out of this view.