Digital Business Ecosystem

Contract n° 507953

Workpackage 17: Composer

Deliverables D17.3: Manual Composer

Information Society Technologies

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: 507953
Project Acronym: DBE
Title: Digital Business Ecosystem

Deliverable N°: 17.3
Due dates: 04/2006
Delivery Date: 05/2006

Short Description:
This document describes the development of the Manual Composer tool and related components for manual service composition within the DBE. This document extends deliverable D17.2 with the latest development efforts in both components, the Manual Composer and the workflow engine. The main advancements since D17.2 are the provision of an intuitive wizard to create a workflow structure, a graphical representation of the workflow, an extension to the PDD editor, and a standards-based workflow engine.

Author: TCD (Trinity College Dublin)
Partners contributed: TCD
Made available to: Public

Versioning		
Version	**Date**	**Author, Organisation**
0.1	21.03.2005	David McKitterick, TCD
0.2	18.04.2005	David McKitterick, TCD Lotte Nickel, TCD
0.3	27.04.2005	Jim Dowling, TCD David McKitterick, TCD Lotte Nickel, TCD
0.4	09.05.2005	David McKitterick, TCD
0.5	10.12.2005	Dominik Dahlem, TCD
0.6	29.12.2005	Dominik Dahlem, TCD
0.7	10.01.2006	Dominik Dahlem, TCD
0.8	01.03.2006	Dominik Dahlem, TCD
0.9	18.04.2006	Dominik Dahlem, TCD
1.0	25.04.2006	Dominik Dahlem, TCD
1.1	11.05.2006	Dominik Dahlem and Rene Meier, TCD

Quality check:

1st **Internal Reviewer** : Pierfranco Ferronato, Soluta

2nd **Internal Reviewer** : Miguel Vidal, SUN

Table of Contents

Illustration Index

Executive Summary

This document describes the design and development of the Manual Composer tool and related components, which when combined enable manual service composition within the DBE. The implementation described in this document was demonstrated as part of the first DBE prototype in the annual review, January 2005, and lead into the final release at the end of April 2006.

The terms ‘service composition’ or ‘workflow’ describe the composition of services as a process flow to provide added-valued composed services. The ambitious goals of service composition involve close integration with the areas of service discovery and service recommendations to enable the creation of static and dynamic workflows. Manual service composition primarily aims to provide users of the DBE, whether they are service developers or just service providers, with a set of intuitive but flexible design tools where they can easily compose and deploy their required business processes. These tools can be used as building blocks for the development of integrated intelligent tools and infrastructure.

The Composer tools described in this document are designed as integrated components within the DBE system architecture and are implemented in a way to allow the necessary interaction with existing and future tools or parts of the DBE infrastructure. The final stage in the development of the Manual Composer tool required a design-time composition language editor based on a workflow specification. Following the common eclipse-based approach for DBE tools, this editor provides the basis for service composition and for further extensions of advanced composition features. The new features include a wizard-like tool to establish the structure of a workflow based on selected service descriptions (SDL), a graphical representation of the workflow, and an extension to the PDD Editor to incorporate the WS-Addressing scheme to identify resources. A core part of the Composer tools is the workflow language specification, where after in-depth research into the area of service composition, the widely deployed Business Process Execution Language was chosen as the language with which to specify workflows. Further, for the purpose of executing workflow descriptions, an open-source engine (ActiveBPEL) is used in order to align the execution environment to existing standards.

The adoption of existing open source software, specifically a workflow engine, was necessary to provide the capability of executing composed services within the DBE Execution Environment. This involved unanticipated work which demonstrates a stable development, deployment, and execution of service compositions within the DBE environment.

1 Introduction

Intelligent service discovery, dynamic service composition and ultimately evolution with the DBE, are all ambitious goals and achieving these requires a basis of sophisticated tools which the Composer work-package is intended to deliver. The tasks involved to deliver such advanced and sophisticated tools is a challenging undertaking but if it is done effectively then it opens the way for an endless potential for additional features and functionality.

The DBE provides a Service-Oriented Architecture (SOA) adapted to the requirements of a SME environment. SOA [21] is a software architecture that models application functionality as modular and loosely coupled services with well defined interfaces. These services are autonomous processing units which interact with their environments through the organised exchange of messages [22]. For DBE providers, SOA is intended to enable them to align their business process to the needs of their customers in an efficient and adaptive manner. In addition to modularity and loose coupling, other attributes of SOAs are coarse grained interfaces, platform interoperability, location transparency, and declarative programming techniques. Effective implementations of process-oriented SOA allows for modularisation of legacy and custom applications and the orchestration of these into flexible business processes. A process-oriented approach to SOA requires a structured model to describe the composition of services.

Service composition can be described using the terms orchestration and choreography. Orchestration [1] refers to an executable business process where the interaction between services at the message level, either internally or externally in relation to organisations, is centrally controlled by a single party. These interactions can result in a long-lived, transactional, multi-step process model. Orchestration can be viewed as the aggregation of existing services to provide a new service. BPEL (Business Process Execution Language) [3] is an example of a commonly used orchestration language. Choreography [1, 20] is a mutual effort which monitors the sequence of messages that may involve multiple partners and multiple sources. Typically this is associated with public message exchanges that occur between multiple external services rather than just a specific business process which is centrally executed by one party. Choreography can be considered more collaborative in nature, where the process is not controlled or specified by one partner. Each partner needs to be aware of the part they will take in the process, the operations to execute, the messages to exchange, and the timing of all these activities. WS-CDL (Web Services Choreography Description Language) [39] is an example of a language for describing choreography.

Orchestration and choreography are not necessarily competing approaches for service composition. Choreography may be viewed as complementary to orchestration where a business process of aggregated services is centrally controlled by a single party while including this process as part of a global viewpoint of peer-to-peer interacting business processes. The combination of these two approaches could aim to provide service composition in a peer-to-peer adaptive environment, such as the DBE. Choreography is at an early stage in its development as a viable approach for service composition within a SME commercial environment, and does not solely provide for the requirements of composing services as part of a business interaction. Orchestration is supported by substantially more emerging standards and tools, and has been chosen as the initial approach for service composition within the DBE because of the manner in which it provides control and flexibility of business process to the SME involved.

There are many technical requirements for service composition which must be addressed when creating tools for designing and executing workflow processes. Such requirements are asynchronous messaging (the independent transfer of messages between services), concurrency (parallel execution of tasks within a process), exception and fault handling (support for when something goes wrong), and flexibility (allowing users to easily make changes to a process). The composition platform must support asynchronous messaging between services to help achieve the reliability, scalability and adaptability needed for long running processes. Concurrent activities within a process in addition to sequential activities are essential to maintain good performance. Exception management and transactional integrity are vital during the execution of long running distributed service compositions. Dynamic and flexible processes can meet the needs of changeable

business environments and competitive markets. Enabling service providers to make on demand changes to the structure or partnerships within their orchestrated workflows is a key requirement for the business and technological advancement of service composition [1].
Given that some architectural decisions in the DBE [36] have followed the SOA approach and some Web Service standards, it was a logical direction for service composition in the DBE to follow existing approaches within the Web Services community. Web Service Composition is an on-going research area with numerous standards, languages and applications emerging from both the Web Services standardisation-community, which is heavily supported by the technology industry, and the Semantic Web research communities, which has strong academic support. The industry sees web services as abstract, standardized interfaces to business processes, while describing these services in WSDL, specifying only the syntax of messages that enter or leave a web service application. Static workflows, which must be manually declared, are then used to describe the order in which messages are to be exchanged [2]. BPEL [3] is becoming the accepted standard language for workflow description. The Semantic Web Community approaches the web service composition problem from a different perspective. In OWL-S (Ontology Web Language for Web Services) Web services are enhanced with semantic descriptions in ontologies that are computer-interpretable, which is an important precondition for automatic discovery, composition and execution of Web services [4]. These approaches and more standards and technologies will be discussed in more detailed in section 2.
The Manual Composer tool is the initial task within the composer's extensible tool set and becomes the major design tool for user managed service composition. The area of service composition is relatively new, with respect to established open standards and integrated development suites, although this area has obtained large research and investment from both industry and research communities. This document is intended to outline the previous and current work in service composition, and remark on how this work in addition to the requirements of this work-package have influenced the design and implementation of the Manual Composer tool and future plans of more advanced tools. The goal of the Manual Composer tool is to enable DBE users to interface with discovery tools to find candidate services, to design service compositions using the discovered services and to publish the newly created compositions as DBE services. The intended users of this graphical tool will initially be DBE developers, but the tool will be refined to enable naïve users, such as DBE providers, to create service compositions using a set of point-and-click features. The Manual Composer tool will interact with structural services and other tools such as the Semantic Registry Service for discovery, the Recommender Service for service suggestions, the Knowledge Base Service for storing of composition models, and the Service Exporter Tool for deployment of executable workflows and publishing of composite services' service manifests (SMs) [37]. The initial feature of the Manual Composer tool is a composition language editor which provides a user with the ability to build and modify service compositions for execution within the DBE. As a composition language task for the DBE was not assigned, it seemed appropriate to select a common established approach which both met the requirements for this task and also the requirements to achieve the overall goals of the project.
The DBE architecture is not based on Web Services technologies, but on the concept of a SOA. Therefore, it has been incorporated into the design of the Manual Composer tool, and related tools, to abstract above any platform specific models, such as Web Services, with the aim of achieving a platform independent modelling (PIM) approach to service composition. Services are described by their Service Manifests (Sms), including their technical interface definition in the form of the Service Definition Language (SDL) [40]. With future additions to the tools, service discovery, selection and inclusion will be a more non-technical task, therefore enabling non-technical users to actively take a role in service composition. The architecture of the DBE describes a component for executing transactional workflows, which corresponds to the task of the Transaction Workflow Manager (TWFM). This task will begin during the second phase of the project and will be a continuation of current work done in relation to workflow executions. The execution tool used with the Manual Composer tool is an adapted implementation of an existing open-source workflow engine, called ActiveBPEL [6]. Extensions have been made to the workflow engine to allow for

protocol independence of service invocations during the execution of service compositions.
The rest of this document explains the area of service composition and the work done towards the goals of this work-package. Section 2 will outline in more detail the background research done in service composition and also describe the adopted technologies which were introduced into the implementation of this task. Section 3 presents the design approach and current implementation of the Manual Composer design tool and how it integrates within the DBE's development suite. Section 4 introduces the ActiveBPEL workflow engine and our advanced modifications to it. Section 5 will review our current plans for further work in service composition and the research done towards the autonomous service composition task, which is due to begin during the second phase of the project.

2 Background and Adopted Technologies

The architectural model of the DBE is closely linked to SOA models and some Web Services standards, so therefore given that there has been extensive activity in service composition from the Web Services community, it seemed beneficial to research this area for possible inclusions to our current work in the Composer work-package. During the initial stages of the Manual Composer task, substantial research was undertaking to assess suitable candidates which could be used as the Manual Composer's composition language. Other research included the areas of service-oriented architectures, workflow orchestration and choreography, composition techniques from the Semantic Web community, and related open source tools with support for service composition.

2.1 Service-Oriented Architecture

"A Service-Oriented Architecture is a component model that inter-relates the different functional units of an application, called services, through well-defined interfaces and contracts between these services." [24] The following diagram, Illustration 1, summarises the SOA approach in a generic fashion. Along with this definition of SOA a definition of a service is also needed. When referring to a service, we define it to be "a collection of protocols and standards used for exchanging data between applications." From this definition we can draw out a definition of many concrete services. For example, a CORBA [25] service would be a service with IIOP over a connection-oriented transport as its protocol and the standards used are those set forth by the object management group [26]. Similarly and with relevance to the 18-month implementation of the DBE a web service is a service with SOAP as its protocol and the standards used are those provided by OASIS [27], W3C [28] and WS-I [29] standards bodies. It should be noted that SOA is an architecture and not a particular implementation of some concrete technology. As such it is feasible and very possible to imagine a SOA-styled system allowing the communication between technologies such as DCOM [30], CORBA and Web Services. This goes against the common misnomer that SOAs are implemented with only Web Service technologies.

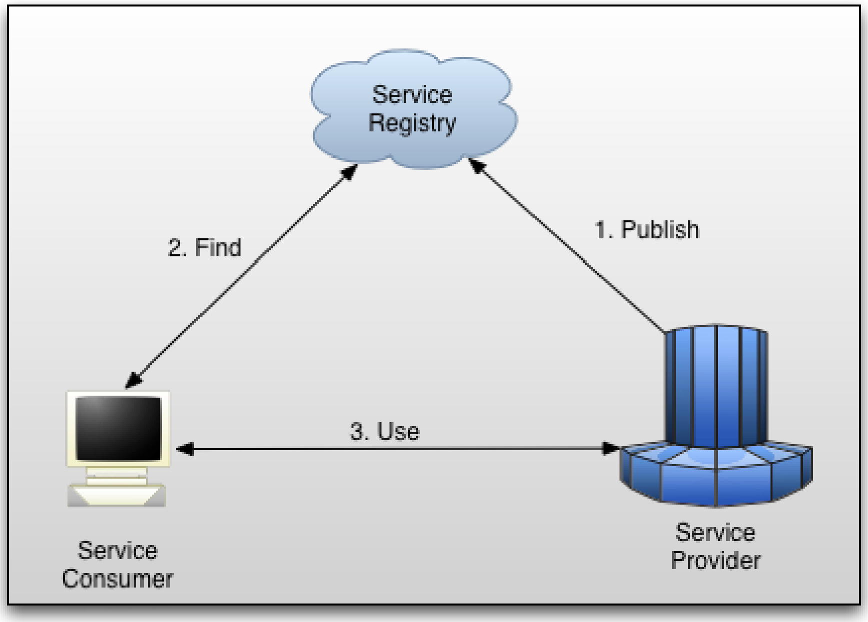


Illustration 1: Generic SOA Model

For a system's architecture to be classified as a SOA the major attribute that they should always exhibit is loose coupling. Loose coupling is achieved when the interfaces used in a service are independent of the hardware platform, the operating system and the programming language that the service is implemented in. As such these interfaces need to be implementation neutral. This allows

services from heterogeneous systems to interact with each other easily and uniformly. Loose coupling gives a service and/or system an elasticity that enables it to survive evolutionary changes in the structure and implementation of the service. It also enables a high-level contractual agreement between services and minimises dependencies between those components. Another desirable and often noted attribute with SOA is the enabling of flexible and late configuration. By allowing services to bind late to each other it is possible to change the configurations at runtime. Consequently, services are not compiled against a specific endpoint of a service for example. This is evident when a look up of a service is performed before invoking it. The dynamic binding of services allows for changes to be made to the service without the worry of notifying dependent consumers of the service about those changes. Finally, for successful SOAs to be realised, services within a SOA should be designed with coarse granularity in mind. The significance of coarse granularity is realised by reducing the dependencies and fewer communication messages between services [31].
Although SOAs are not necessarily implemented with web service technologies, this has been the most common and visible architectural style to date. For a typical web service based SOA to be realised there are a number of basic technologies that are required to be utilised, such as UDDI (Universal Description, Discovery, and Integration), SOAP (Simple Object Access Protocol) and HTTP (Hyper-Text Transfer Protocol). There are many reasons for using web services in a SOA. Some of which are given below:

- A major advantage of web services is that by using HTTP, they can work through many common firewall security measures without requiring changes to their filtering rules. This has been the downfall of many architectures for use over large public networks. Examples include DCOM and CORBA.

- Due to SOA’s property of loose coupling, web services allow the reuse of services and components within an infrastructure with minimal dependencies.

- Consequentially, Web Services allow services from different companies and locations to be combined easily to provide an integrated composed service. An example of this is using the Business Process Execution Language (BPEL) [3]. This is due to the loose coupling and dependency minimising nature of SOAs as discussed previously.

- As Web Services leverage open standards and protocols, this provides interoperability between various software applications running on various platforms and architectures. Also as many of the protocols and data formats are text based, it makes it easy for developers to understand and debug request/response flows between services.

- Web Services add an abstraction layer to the underlying middleware paradigm. It allows for transparent execution of J2EE, CORBA, or message-oriented middleware.

Although web service would appear to be a “silver bullet” cure they do have their disadvantages. Web services will suffer performance penalties when compared to older and more established technologies such as CORBA and DCOM. This performance penalty mainly is due to the goal of achieving interoperability between technologies and standards. Also, problems arise when the WSDL is fully qualified with port and endpoint information. In the WSDL definition, the developer of the service may need to enter the well-known URL of where the web service will be made available. When this service is exposed as a Web Service, the WSDL is advertised through the UDDI registry. With the web service implemented and deployed, the developer now needs to make the new service accessible and advertised to service consumers. The WSDL description of the new service, containing the static, well-known endpoint, is then registered to the UDDI registry where a copy of the service’s WSDL is kept for future requests by service consumers. The problem arises if the service provider cannot guarantee that the service’s endpoint will remain constant and static. This will affect service consumers who try to access the web service whose address is changing so

often as to render the updating of its WSDL impractical. This problem then becomes a barrier to potential service providers who cannot acquire the necessary resources, be they technical or financial, to provide a static and well-known URL for their services. This can be envisioned as a potential problem for SMEs trying to bridge the "digital divide". Howeverr, the DBE implemented means of allowing a pure late binding to a service, especially taking into account the dynamic nature of the ecosystem.

2.2 Current Research and Composition Languages

2.2.1 BPEL

The Business Process Execution Language for Web Services (BPEL or BPEL4WS) [3] is a workflow orchestration language which defines a process-centric model for the formal specification of the behaviour of business processes based on the interaction of the process and it partners [2]. BPEL represents the combination of earlier workflow languages, WSFL (Web Services Flow Language) and XLANG, into one cohesive package defined in XML that supports two kinds of business processes, executable and abstract ones. WSFL was created by IBM and is designed to support for graphical oriented processes. XLANG was created by Microsoft and is a block-structured language with structural constructs for processes. The latest version, BPEL4WS 1.1, is layered on top of other Web technologies such as WSDL 1.1, XML Schema 1.0, XPath 1.0, and WS Addressing. BPEL4WS has been submitted to the standardization consortium, OASIS, where the WS-BPEL TC (Web Services Business Process Execution Language Technical Committee) has been formed [23]. BPEL provides programming like constructs (sequence, switch, while, pick) as well as graph-based links that represent additional ordering constraints on the constructs. The BPEL notation includes flow control, variables, concurrent execution, input and output, transaction scoping/compensation, and error handling [2]. Currently, there exists various implementations including engines and editors for BPEL, but there are some open source projects which provide a BPEL engine. These will be discussed in more detail under the adopted technologies section. The BPEL language and its implementations will be discussed in more detail in Chapters 3 and 4.

2.2.2 WS-CDL

The Web Services Choreography Description Language (WS-CDL) [39], as defined by the W3C Choreography working group [32], is used to model multi-party collaborations which describe the externally observable behavior of peer services and their clients by specifying the message exchanges between them. Choreography may be seen as complementary to the orchestration of services, where it describes the relationships between services in a peer-to-peer collaboration without any central orchestration of the services involved [33, 34]. Therefore WS-CDL may be viewed as complementary to BPEL and not intended for the same purpose. BPEL could model the business process of internal services which is then included in a global external viewpoint of peer-to-peer interacting business processes defined by WS-CDL.
A WS-CDL choreography description is a container for a set of activities which are to be performed by the partners involved. The activities are categorized under control-flow, work-unit and basic activities. The control-flow activities provide a block structure with the sequence, parallel and choice activities. A work-unit activity provides conditional attributes to the collaboration. Basic activities provide interaction and data manipulation activities. One or more WS-CDL descriptions can be contained within a package entity, which holds all common information for all choreographies contained in the package. A package also describes RoleTypes that exhibit behaviors and RelationshipTypes that occur between two RoleTypes. ParticipantTypes may also be contained in the package which describe logical grouping of roles [33].
Choreography, and in particular WS-CDL, seem to be a promising approach for the creation of decentralized peer-to-peer collaborations between services and even orchestrated business processes, but it may be that the WS-CDL standardization effort has come too early in the evolution of SOAs. Unlike BPEL, which was created from two existing languages, WS-CDL was created without been

derived form any prior work. BPEL is gaining popularity, as seen by the number of tools available which support it, while WS-CDL still remains an immature specification.

2.2.3 OWL-S

As part of the DARPA Agent Markup Language program (DAML) the Semantic Web community defines OWL-S (formerly DAML-S) [4]. OWL-S is an OWL-based Web service ontology with three interrelated sub-ontologies, known as the service profile, service model and service grounding.
A service profile is used to advertise “what a service does”. The profile specifies the inputs, outputs, preconditions and effects (IOPE) of a service as well as additional information such as the service provider, quality of service and constraints. The profile is only used to advertise a service in a registry in a machine-readable way. Once a service is selected the profile becomes useless and can be discarded.
A service model is used to describe “how a service works” viewing the service as a process. The process model is independent of any execution logic. OWL-S differentiates between atomic processes, simple processes and composite processes. Atomic processes can be directly invoked and executed in a single interaction. Composite processes are composed of atomic or other composite processes. The composition can be defined by control constructs (if-then-else, sequence, split and join). A simple process is an abstract process and cannot be invoked. It is used to provide an alternative view of an atomic process or an abstraction of a composite process. OWL-S was designed to be agnostic with respect to a process model formalism in order to remain compatible with emerging standards for process modeling like BPEL.
A service grounding specifies “how a service can be accessed”. Both the service profile and service model are abstract representations and the service grounding is responsible for mapping the abstract inputs and outputs of processes to concrete WSDL messages. Each atomic process must be provided with a service grounding. There can be more than one grounding for a service. The advantages of OWL-S are the rich semantics that enable automatic discovery, selection and composition. While the main focus of BPEL is to provide an executable service composition the goal of OWL-S reaches much further into the intelligent and autonomous process area. The disadvantage of OWL-S is that there currently are no complete execution tools available and there is a lack of support from industry partners.

2.2.4 Meteor-S

The interesting possibility of merging the rich semantic descriptions and ontologies that OWL-S offers with the BPEL workflow descriptions is already researched and adopted by the METEOR-S project of the LSDIS Lab, University of Georgia. The METEOR-S framework aims to support the complete lifecycle of semantic Web processes [5]. The lifecycle is described by the stages annotation, publication, discovery, selection, composition and execution. The framework can be split up into a front-end and back-end part (see Illustration 3). The front-end is responsible for the stages annotation and publication of semantic Web services, whereas the back-end covers discovery, selection, composition and execution of semantic Web services.
The front-end [17] consists of the components Semantic Web Service Designer, Semantic Description Generator and Publishing Interface. The Semantic Web Service Designer is used to annotate Web service descriptions semantically. The annotated service description is passed on to the Semantic Description Generator. There is currently no standard for semantically described WSDL services. Therefore the Semantic Description Generator generates annotated WSDL 1.1 (semantic features added via permissible extensibility elements so that file adheres to current industry standard), WSDL-S (semantically enriched WSDL 2.0 document with extensions that were proposed by METEOR-S) and OWL-S files (profile, grounding and partial process model) in order to incorporate different approaches. The Publishing Interface is responsible for publishing the semantic descriptions of Web services in an UDDI registry [19] that was enhanced for semantic publication and discovery of Web services.

The back-end [16] consists of the components Abstract Process Designer, Discovery Engine, Constraint Analyzer and Binder. The Abstract Process Designer is used to create an abstract process in BPEL. The required Web services are described in service templates that enable to either bind to a known service manually or specify a semantic description of the required service for automatic discovery in the Discovery Engine and selection of the "best" available service in the Constraint Analyzer. The Discovery Engine uses the service templates to query the enhanced UDDI registry for matching service descriptions. These service descriptions are passed to the Constraint Analyzer. The service templates contain the user's preferences for partners, QoS requirements and constraints like service dependencies that are used to find the "best" matching services. The Binder is finally responsible for the late binding of services to generate an executable BPEL process.

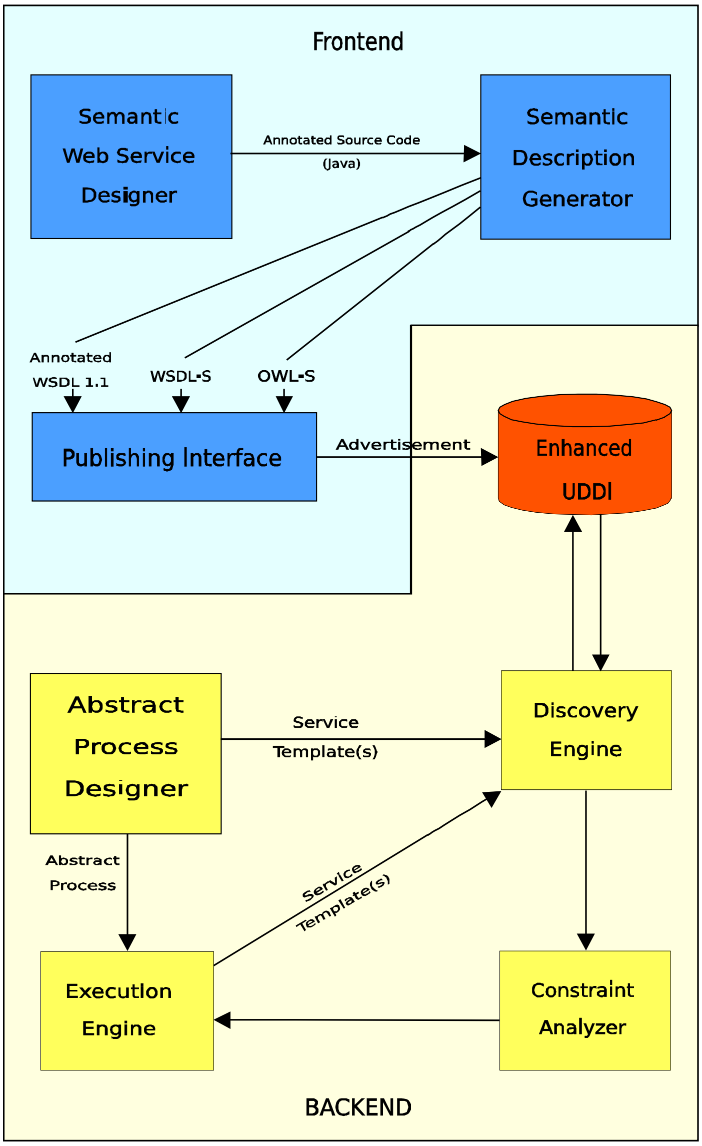


Illustration 2: Meteor-S Architecture

2.2.4.1 SDS (Semantic Discovery Service)

Another approach to combine BPEL and OWL-S to overcome the static definition of service partners in a BPEL workflow comes from some authors of the OWL-S research community [15]. They introduce a Semantic Discovery Service (SDS) that interposes between the BPWS4J engine and service partners. Instead of invoking requests on statically bound partners the BPWS4J engine routes requests to the SDS. Service partners are described, advertised and discovered via a OWL-S service profile. They adopted the DAML Query Language (DQL) as the language and protocol to query a knowledge base (KB) of OWL-S service profiles. The SDS constructs a DQL query and

sends it to the DQL server. The DQL server uses the Java Theorem Power (JTP) DAML + OIL reasoner to find service profiles in the KB that match the query. Matching service profiles are returned to the SDS in answer bundles. The SDS chooses a matching service partner and invokes the partner's endpoint. Upon receiving a reply from the partner the SDS forwards the response to the BPWS4J engine. This interaction flow is stateless (no knowledge about prior interaction) and agnostic to content of service descriptions and invocation messages.

The authors describe their approach as a way to enable BPWS4J with automated Web service discovery but clearly state that from their point of view this is not equivalent to automated Web service composition. In their opinion, automated Web service composition requires that a workflow composes itself at runtime from a set of inputs and outputs as well as taking preconditions and effects into account.

2.2.5 AOBPEL

AO4BPEL [14] is an approach to merge aspect-oriented programming (AOP) with BPEL in order to solve the two problems crosscutting concerns (like monitoring, accounting or logging) and dynamic composition. BPWS4J (IBM's BPEL engine) is extended to enable (un)plug aspects into the composition logic at runtime. At specific join points (points in the execution of a program) the BPEL engine checks if aspects have to be executed. In this case the BPEL engine hands over to the Aspect Manager, which executes the respective aspect. An advice specifies when (before, after or instead) and what code has to be executed when a join point is reached. The activities invoke and reply are used as pointcuts (pattern by which a join point is identified) in the current approach as they are the main interaction points with partners. Interesting about this approach is that the BPEL workflow description remains the same and thereby compatible with other BPEL engines.

2.2.6 Self-Serv

Another approach to the service composition problem is the Self-Serv framework [13]. Self-Serv introduces a P2P-based orchestration model approach to the Web service composition problem. Self-Serv is based on two major concepts: composite services and service containers.
A composite service is composed of several other composite or elementary services. The business logic of a composite service is described in state charts. State charts were chosen because they possess all the formal semantics, are well-known and well-supported and can be easily transformed into other process modeling languages like BPEL. A service container is a service that aggregates several substitutable services. The aggregated services do not have to implement the same interface but can provide a mapping from the general service interface that a container supports and to their proprietary service interface. The container enables to measure the quality of service for each offered service and perform load-balancing between the offered services. The service requests can be routed to the best matching, available and not overused service.
The P2P orchestration is based on state coordinators and routing tables. Self-Serv generates one state coordinator for each state in the state chart of a composite service. Each state coordinator is associated with a routing table that defines preconditions that have to be met for a state transition. The initial state coordinator initiates the composition and collects the results from the other states but is not responsible for state transitions. The composition is thereby self-orchestrated and not centralized.

2.3 Existing Tools and Technologies

The primary requirement of the Manual Composer task was to provide a tool from which a DBE user could create service compositions, but a model of a service composition is useless without any infrastructure to execute this model as a workflow. An adopted workflow engine would have to be extensible to the emerging architecture of the DBE and also compatible with the composition meta-model being used by the Manual Composer tool. During the research stage of this work-package, it became evident there were limited options for a suitable workflow engine, as the selected

composition language, BPEL, was an immature language with mostly industry support and proprietary implementations, such as IBM's BPWS4J workflow engine. Given that the DBE project is an open source software project, our research resulted in the options of only two open source projects which at that time provided suitable workflow engines, ActiveBPEL and Twister. Other technologies used in the implementation of this task were the Eclipse platform and frameworks.

2.3.1 ActiveBPEL Engine

The ActiveBPEL [6] engine is an open source implementation of a BPEL engine, written in Java. It reads BPEL process definitions (and other inputs such as WSDL files) and creates representations of BPEL processes. When an incoming message triggers a start activity, the engine creates a new process instance and starts it running. The engine takes care of persistence, queues, alarms, and many other execution details. This engine runs on Tomcat and using Axis for web service executions. The current release is version 2.0 and it implements the BPEL4WS specification, version 1.1 [3]. The creators of ActiveBPEL are connected with the WSBPEL technical committee and therefore they will update their implementation following any specification changes. See Illustration 3 for the architecture of the ActiveBPEL engine.

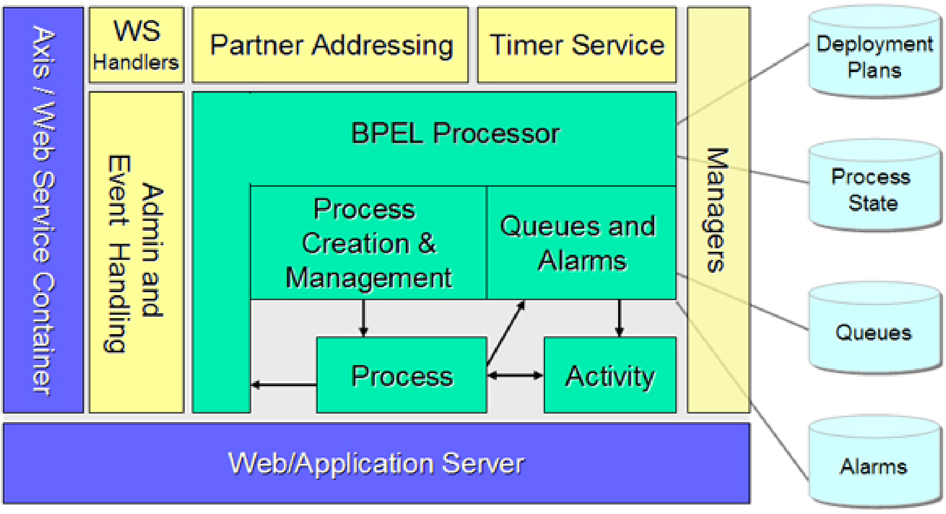


Illustration 3: ActiveBPEL Engine Architecture

The ActiveBPEL engine provides a good robust implementation which ideally suits the extensible needs of the project, although as BPEL was originally designed for the interaction between web services, the engine was implemented with Axis as its web service container and SOAP as its communication protocol. Fortunately, the pluggable architecture of ActiveBPEL's engine makes it possible to modify the engine to use any protocol or even any platform, which is favourable towards the goals of this work-package.

2.3.2 BPWS4J Engine

IBM's Business Process Execution Language for Web Services Java Run-Time (BPWS4J) [35] provides a platform for creating and executing business processes written in BPEL. The main component of this platform is a BPEL workflow engine which uses Apache Axis for incoming web service requests and is integrated with Websphere Application server and the Apache Tomcat servlet container. Other tools under this platform include a basic eclipse-based editor for creating BPEL models and a tool that validates BPEL documents. As a requirement for any software included in the DBE project, the workflow engine should have an open source license, but this was not the case for IBM's BPEL engine.

2.3.3 Twister WS-BPEL Engine

Another available open source workflow engine was provided by the Twister [8] project. Twister is

a WS-BPEL (an OASIS continuation of BPEL4WS 1.0) compliant business process engine, providing web services invocation and direct end-user interaction using a work list. It lets you deploy your processes described in WS-BPEL in the engine that will execute them. It will acknowledge the messages you send and produce new messages invoking any service. The latest release of this engine is version 0.3. In the meantime, Twister has been donated to the Apache Software Foundation and has been renamed to Agila. At the moment the project is waiting for the requirements of the incubation into the Jakarta project to be completed.

2.3.4 FADA

The Federated Advanced Directory Architecture (FADA) [41] is a distributed directory of services, but appears to a client like a unique virtual lookup server. It is written in Java, and based on the concepts of the Jini Networking Technology. The main difference between FADA and Jini, is that with Jini you can have as many lookup servers as you want within a LAN, but the client is responsible to find all these servers separately. In Jini, lookup servers do not cooperate, so in order to find a service proxy the client has to ask all of them. Broadcast protocols are fairly efficient in a LAN, so a client can make just one lookup request on all lookup servers, although this can not be achieved in the Internet without additional logic and cooperation among lookup servers. FADA was designed to address this issue without a central authority or common communication channel, and lookup servers internal to the overall virtual lookup server operation of FADA work in a peer-to-peer fashion. Essentially, FADA is an extended version of Jini designed for the Internet, although scalability issues exist given that FADA only performs well with approximately one hundred nodes due to the broadcast mechanism adopted for peer-to-peer communications. The overall topology is a hybrid centralized/decentralized topology, where the distributed FADA architecture acts as a centralized server from the clients' point of view. FADA (as well as Jini) holds proxies for services, i.e. a Service Proxy. A proxy is a Java class that performs communication with a real service, and that can be downloaded at run-time by clients. Clients use the public methods on the proxies to access the services. These public methods are specified in Java interfaces that service proxies implement.

2.3.5 Eclipse Platform and Frameworks

The Eclipse [9] project provides an open extensible IDE which has been adopted by the DBE project as the basis for the first prototype implementation. This extensible environment allows the project's developers to create pluggable tools, such as model editors and graphical viewers, which can easily be integrated together as a unique development environment for the DBE users. In addition to the Java development tools, which are available as standard, the combination of the 'DBE Studio' tools provide a platform, a Service Factory, where DBE users (mainly DBE Developers) can perform various tasks. Such tasks would be to create business and service models, generate and implement service components, interact with DBE structural services, and deploy services to the DBE's Execution Environment. Also, the DBE Studio contains the Manual Composer tool where users can design and deploy service compositions.
The eclipse platform provides some useful frameworks from which tools can be easily created and adapted. Such frameworks from the Eclipse Tools project which were used in the development of the Manual Composer tool are GEF (Graphical Editing Framework) and EMF (Eclipse Modelling Framework).

2.3.5.1 GEF (Graphical Editing Framework)

GEF [11] allows developers to easily create a rich graphical editor providing representations for existing models. GEF uses the SWT (Standard Widget Toolkit)-based drawing plug-in, Draw2d, to create a graphical environment within Eclipse. GEF employs an MVC (model-view-controller) architecture which enables simple changes to be applied to the model from the view. GEF is completely application neutral and provides the groundwork to build almost any application for any model. Such editors may include flow builders, GUI builders, UML diagram editors (such as

activity and class modeling diagrams), and WYSIWYG text editors like HTML.

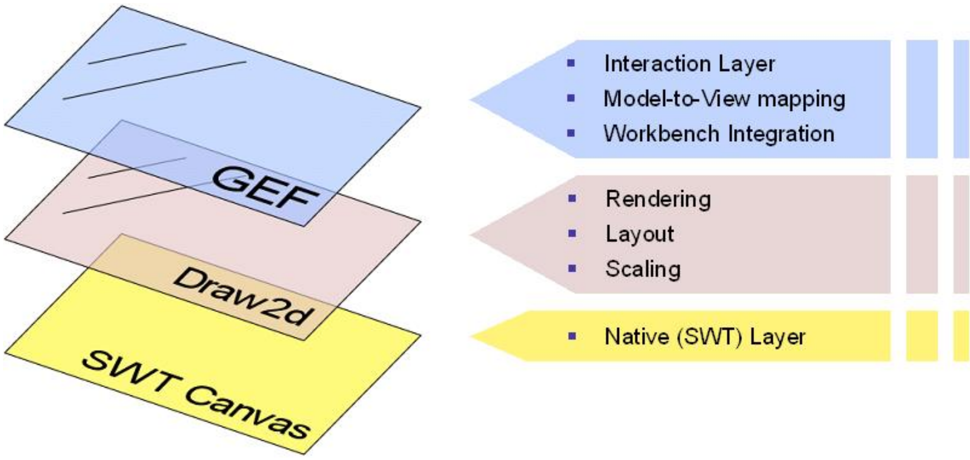


Illustration 4: Eclipse Graphical Layers

2.3.5.2 EMF (Eclipse Modelling Framework)

EMF [10] is a Java modelling framework for building tools and other applications based on a structured data model. The framework was designed to ease the design and implementation of a structured model. The framework provides a code generation facility to produce a set of Java classes which represent the model, which was original specified in XMI, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or UML modelling tools like Rational Rose, then imported into EMF as an XMI model. EMF follows the Model Driven Architecture (MDA), as it started as a Meta Object Facility (MOF), of the Object Management Group (OMG), implementation but has now evolved to an enhancement of MOF2.0.

3 Manual Composition and Design-Time Tools

3.1 Use Cases for Manual Composition

The following use cases introduce the functional model of the Manual Composer tool and the related tools it uses to perform the task of manual composition. Illustration 5 shows the use cases for some of the tools in the Service Factory (DBE Studio) [36] including the Manual Composer tool.

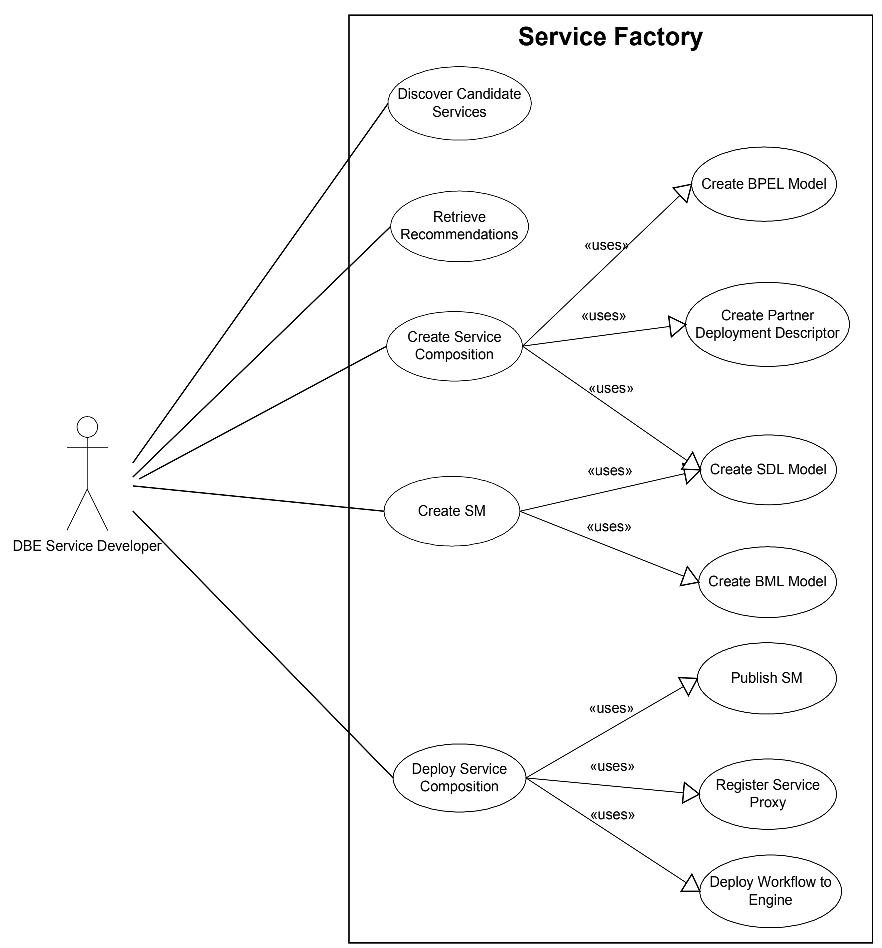


Illustration 5: Manual Composition Use Cases

The following tables describe each of the use cases and put them into the context of developing and deploying a composed DBE service. Some the use cases described in this section (i.e. 'Discover Candidate Services', 'Retrieve Recommendations' and 'Create SM') are not exclusively use cases for manual composition but they are dependent cases and play an important role in the whole process of composing services. For a more complete definition of these particular use cases, see the referenced documentation by the relevant responsible partners.

Use Case	*Discover Candidate Services*
Area:	Service Factory (Query Formulator and Semantic Discovery Tool)
Objective:	Find existing DBE Services which are suitable to be used within a new composition
Actors:	Developer

Use Case	*Discover Candidate Services*
Pre-Conditions:	• Some Services are published to the Semantic Registry and registered to Fada
Post-Conditions:	• The developer can require the SM of any discovered service, the therefore access the encapsulated SDL and BML models
Description:	The developer will use the Query Formulator and Semantic Discovery Tool [37, 38] to find existing published SMs for services which relate to an entered semantic description and a syntactic interface definition. See other documentation [37, 38] for more in-depth descriptions of this use case.
Dependencies:	Query Formulator and Semantic Discovery Tool, Semantic Registry Service
Responsible Partner:	TUC

Use Case	*Retrieve Recommendations (N/A for first 18th month implementation)*
Area:	Service Factory (Recommender Tool)
Objective:	Use the Recommender Tool [38] to obtain recommendations of existing DBE Services.
Actors:	Developer
Pre-Conditions:	• Some Services are published to the Semantic Registry and registered to Fada
Post-Conditions:	• The developer will have a list of recommended services
Description:	The developer will use the Recommender Tool for retrieve intelligent recommendations of existing DBE Services relating to specific service or business requirements. See other documentation [38] for more in-depth descriptions of this use case.
Dependencies:	Recommender Tool, Semantic Registry Service
Responsible Partner:	TUC

Use Case	*Create Service Composition*
Area:	Service Factory (Manual Composer Tool)
Objective:	Create a Workflow Model and related files.
Actors:	Developer
Pre-Conditions:	• Existing DBE Services have been discovered or recommended for use in a composition

Use Case	*Create Service Composition*
Post-Conditions:	• The service composition can be deployed and executed on the workflow engine
Description:	The developer will create the executable business process using the graphical BPEL editor, the Partner Deployment Descriptor editor, and the SDL editor.
Dependencies:	Create BPEL Model, Create Partner Deployment Descriptor, Create SDL Model
Responsible Partner:	TCD

Use Case	*Create Service Manifest*
Area:	Service Factory (Service Manifest Creator)
Objective:	Create/Modify the Service Manifest of a DBE Service
Actors:	Developer
Pre-Conditions:	• BML model must be created • SDL model must be created
Post-Conditions:	• If the Service Manifest is published, it becomes available to DBE participants. But the corresponding service can not be executed. • If the Service Manifest is published, the developer is owner of this Service Manifest.
Description:	The developer may create the Service Manifest once the BML and the SDL model have been created. The Service Manifest can be stored in the semantic registry at this point. See other documentation [36] for more in-depth descriptions of this use case.
Dependencies:	Service Manifest Creator Tool
Responsible Partner:	Soluta

Use Case	*Deploy Service Composition*
Area:	Service Factory (Service Exporter Tool)
Objective:	Deploy the composed DBE Service
Actors:	Developer
Pre-Conditions:	• BML model must be created • SDL model must be created • SM must be created • BPEL model must be created

Use Case	*Deploy Service Composition*
	• Partner Deployment Descriptor must be created
Post-Conditions:	• The service is owned by the developer and can not be changed by anyone else in the DBE. • The composed service is available for execution and its corresponding models can be browsed in the Semantic Registry.
Description:	The developer will deploy a composed DBE Service. The deployment includes the registration of the DBE (composed) Service with FADA, the publication of the Service Manifest in the Semantic Registry and the deployment of the BPEL model (with related files) to the workflow engine. As a result, the composed service is visible to DBE consumers as a DBE service and therefore can be utilised.
Dependencies:	Register DBE Service, Publish Service Manifest, Deploy Workflow to Engine
Responsible Partner:	TCD

3.2 BPEL: A Composition Language

BPEL4WS 1.1 is the latest specification and only the second version released for this composition language. The specification outlines all the main attributes of this meta-model, and also two usage scenarios, one for implementing executable business processes and one for describing non-executable abstract processes. Executable business processes model actual behavior of a participant in a business interaction specifying the exact details of the business process. These processes can be executed by a BPEL workflow engine. Abstract processes, or also called abstract business protocols, specify the public message exchange behavior of each of the parties involved in the protocol, without exposing their internal behavior. Abstract processes cannot be executed. BPEL is intended to be used to model the behavior of both executable and abstract processes [3, 20].
The main part of any BPEL model is a process element. A BPEL process specifies the actual order in which partner services are invoked. All BPEL models must have at least one process. A process may consist of the following elements [3, 7, 20]:

- **Partner links**: These describe the relationship between two services at the interface level. Partner links provide a link to services that are being invoked by the process, but also to clients which are invoking the process.
- **Partners**: A Partner can describe any party taking part in the process, either invoked services or clients of the process.
- **Variables**: These are containers for values and provide a means for holding messages that constitute the state of the process.
- **Correlation Sets**: These are named groups of properties that uniquely identify the business process. At different times in the process, different correlation sets may identify the process.
- **Fault handlers**: These are attached to a scope activity to provide a way define a set of custom fault-handling activities, to describe what to do when a fault or error occurs during the execution of a process.

- **Compensation handlers**: Provides a wrapper for a compensation activity which may describe how to reverse or compensate already-completed business processes.
- **Event handlers**: Triggers an enclosed activity or set of activities after an event occurs due to either incoming messages or alarms.
- **Parent Activity**: A single BPEL activity, usually a container for other activities, e.g. a sequence. This initial activity (or there maybe many parent activities) can contain an extensible set of activities which provide the structure and logic for the business process.

Activities are the building blocks of BPEL processes. Basic activities represent basic constructs and are used to define simple behavior like receiving a message, invoking a service, generating a response for synchronous operations and assigning values to variables. Structured activities are similar to conditional and looping constructs in programming languages and are used to combine the basic activities of the process. Additional activities introduce variable scoping and handle abnormal activities such as process termination and compensation. Activities are joined by links, either explicit or implicit. The path taken through the activities and links is determined by many factors, including the values of variables and the evaluation of expressions. Scopes provide a behavioral context for each activity and are similar to programming language blocks that introduce new variable scope and exception handling mechanisms. A Scope can provide event handlers, fault handlers, a compensation handler, date variables and correlation sets. Some activities such as Scope and Invoke generate new scopes, whether implicitly or explicitly [7, 3].
A process can be defined in a synchronous or asynchronous manner. A synchronous process is when the process will block a client, after a receive activity was triggered, until the process is completed and it returns a result to the client via the reply activity. This approach is only suitable for short lasting processes, as a connection to a client may time out during a longer process. An asynchronous process is when the process does not block the client and uses a call-back mechanism to return the result back to the client. This would usually involve the process performing an invoke activity on the client. This approach is better for longer lasting processes, which could take days or weeks to complete, perhaps due to some necessary human interaction during the execution of the process [20].
The following tables show the main activities specified by the BPEL meta-model [7]:

Basic Activities	***Description***
Receive	Block and wait for a message from a partner (client or invoked service). A receive activity is also used to instantiate the business process, if the createInstance attribute is set to yes. A receive activity with the attribute set to yes must be an initial activity in the process.
Reply	Used to send a synchronous response to a request previously accepted through a receive activity.
Invoke	Invoking another service (partner) which can be either a synchronous request-response or an asynchronous one-way operation.
Assign	Used to copy data from one variable to another or to set new data to a variable.
Throw	Generate a fault
Wait	Wait for a given time period (time-out) or until a particular time has passed (alarm)
Empty	An empty activity

Structured Activities	***Description***
Sequence	Contains one or more activities which are executed sequentially in the order in

	which they are listed.
Switch	Supports conditional behaviour just like a "switch" or "case" statement in other programming languages.
While	Supports repeated execution of an iterative activity while a condition remains Boolean true.
Pick	Blocks and waits for one of a set of events to occur, then executes the activity associated with the event that has occurred.
Flow	Provides concurrency and synchronization, where activities are executed in concurrently and links can provide synchronization.

Additional Activities	***Description***
Scope	Provides a behavioural context for activities, and can contain event handlers, fault handlers, a compensation handler, date variables and correlation sets.
Compensate	Invoke compensation on an inner scope that has already completed normally
Terminate	Immediately terminate a business process instance
OnMessage	Triggers an event when a message arrives
OnAlarm	Triggers a time-out event. The clock for the duration starts at the point in time when the associate scope begins.

See Appendix A for an extended view of the BPEL meta-model and Section 3.5 for a BPEL process example.

3.3 BPEL Manual Composer Tool

The design of the BPEL Manual Composer tool centres on a graphical editor and a composition wizard for this composition language. This editor is the core component as it allows the user to graphically design the composed service as a workflow process, while the wizard uses simple rules to help a user to select services and create model structures. The implementation of both the editor and the wizard fully support the BPEL meta-model previously discussed in section 3.2. The design of the editor provides a 3-view editor where each view has a more abstracted representation of the BPEL model. The intention was to provide two levels of graphical abstraction and granularity to suit both a semi-technical user and a BPEL developer, where the wizard and the graphical editor attempt to address the needs of both user types respectively.
To further extend to the functionality of this tool, integration with other tools and core structural services. For service discovery, both the editor and the wizard have been integrated with the Query Formulator and Semantic Discovery Tool to enable the discovery of DBE services based on SDL, which is explained in more detail in section 3.6. For deployment of a composed workflow as a service within the DBE, the Service Exporter Tool has been extended to enable workflow deployment to the Servent. As each service composition from the perspective of a client is an atomic DBE service, and then the composed service requires a Service Manifest (SM), which is then published to the Semantic Registry during the exporting process (publishing and/or deployment). Using the Servent's adapter and filtering mechanism, a BPEL composition is deployed using a generic BPEL adapter class. Therefore, when a client request for this service is received into the Servent, a registered BPEL filter redirects this request to the BPEL engine.
The Manual Composer tool is implemented as an eclipse plug-in and is integrated within the DBE Studio. The eclipse plug-in consists of two graphical editors, a BPEL editor and a PDD editor, and a composition wizard. The SDL editor was developed by Soluta and integrated within the DBE Studio as an eclipse plug-in.

3.4 BPEL Graphical Editor

The main feature of the BPEL Manual Composer Tool is a BPEL graphical editor. This editor provides three views: a BPEL process overview, a graphical tree editing view, and a source editing view. The BPEL process overview is first visible view when the editor is opened. This view provides a graphical summary of the BPEL process and its internal structure and activities. It also shows a list of the partners/services included in the composition. To the left-hand side of this view is a task palette where additional operations can be performed. These tasks include service discovery, SDL and PDD file generation and BPEL validation. The individual tasks can be accessed by clicking on the task button and then clicking anywhere in the 'Process Structure & Description' box. This process view, which is the higher abstraction level GUI, is aimed at the semi-technical user and when used in conjunction with the composition wizard it will provide this user with the functionality necessary to create simple structured service compositions without having in-depth knowledge of BPEL. Of course by abstracting some of the process attributes within the composition model, it is inevitable that some of the power and flexibility provided by the BPEL meta-model will have to be generalised.
The second level of abstraction, the graphical tree editing view, enables a developer with in-depth knowledge of BPEL to create, edit and remove various BPEL activities within a process, while adapting the structure and behaviour of the service composition. This view has also graphical operations, such as a drag and drop mechanism of model objectives. This view is significantly more complex than the process view (and composition wizard) so the user will need a good standard of knowledge of the BPEL meta-model and all its attributes. In addition, there is a third view, the XML source editing view, which will allow textual editing of BPEL source files.

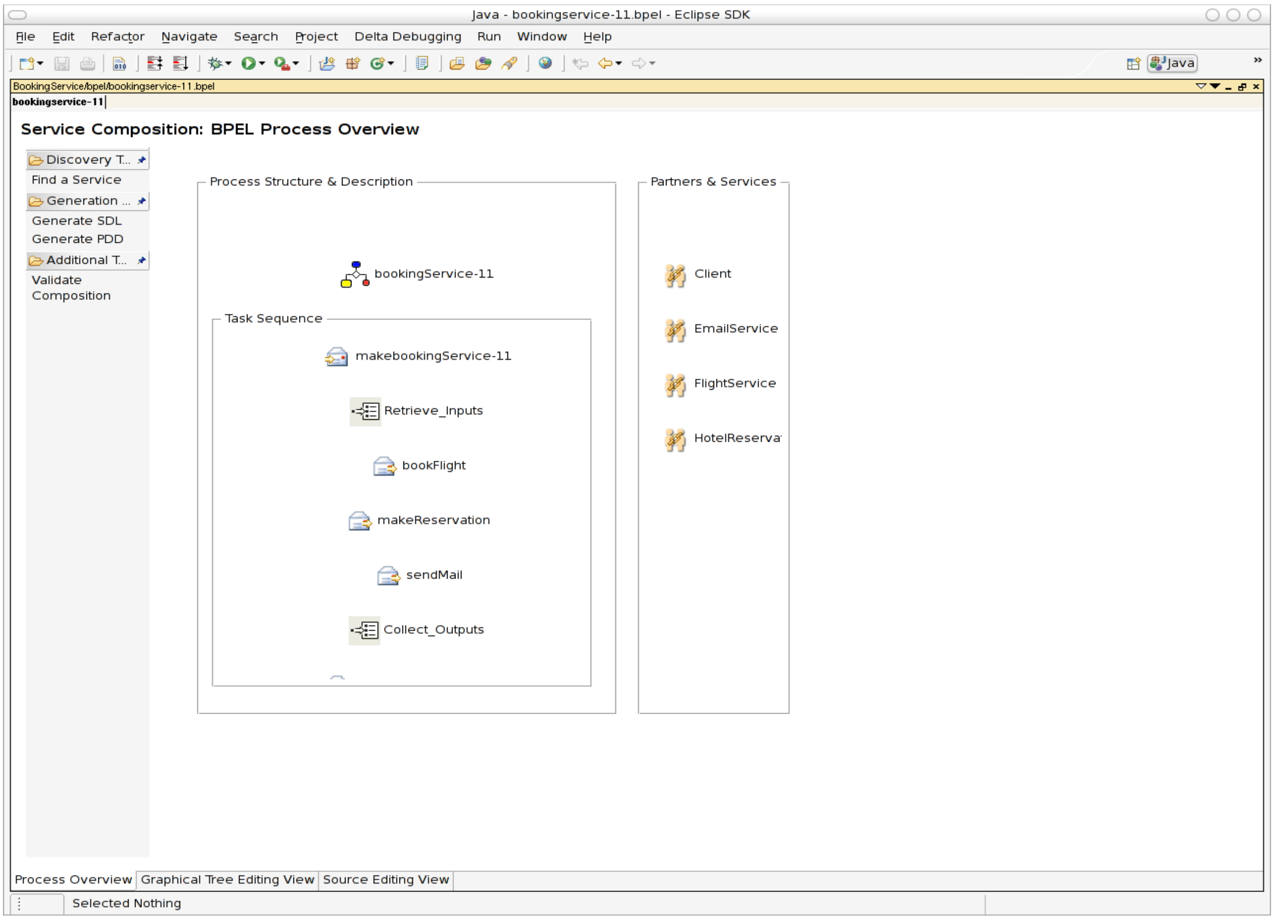


Illustration 6: BPEL Graphical Editor

The graphical editor is also interlinked with other editors for which to provide more concrete model data about the services involved within the workflow process, e.g. service reference and workflow engine specific deployment information. Each atomic service will have a SDL description which

can be referenced from the BPEL description. Following adaptations to an open source BPEL workflow engine, services can be executed based on their SDL definition, therefore not requiring a dependency on concrete binding information. The adapted workflow engine requires a PDD (Partner Deployment Descriptor) file for deployment of a BPEL model. This file tells the engine about your deployment with references to the declared *Partnerlinks* and to the SDL definitions of the services included in the composition.
Illustration 6 shows a sample process view of the first released version of the BPEL editor. To assist in the development of this editor, eclipse's graphical editing framework (GEF) was adopted as the graphical development platform, given its close links to eclipse's plug-in development environment. Following the MDA approach adopted in the project, the eclipse modelling framework (EMF) was used to assist the creation of a BPEL meta-model and basic editor constructs in Java. After some modifications to the EMF assisted code, a basic graphical BPEL editor was added to the Manual Composer plug-in, as shown in Illustration 7.

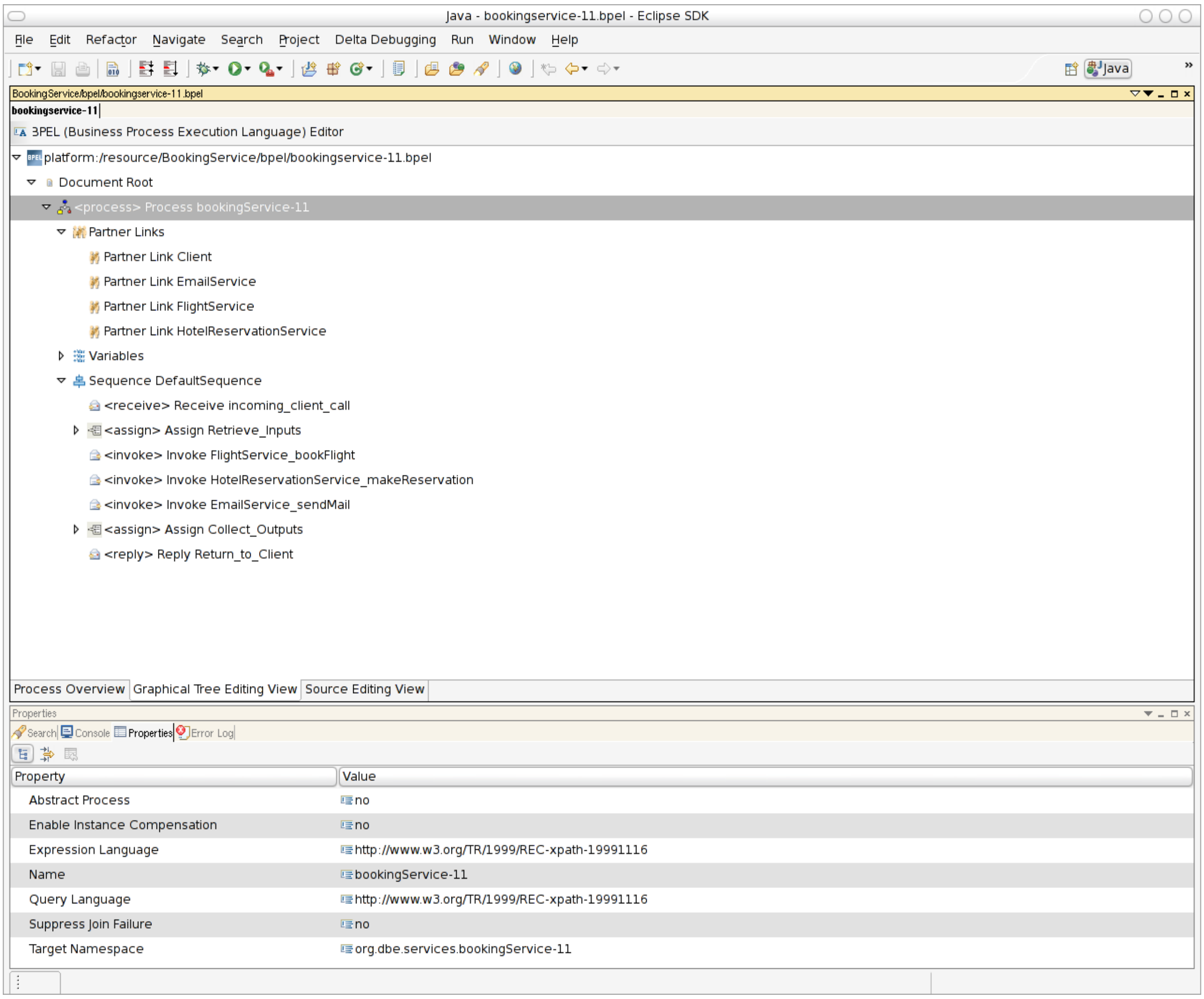


Illustration 7: BPEL Editor

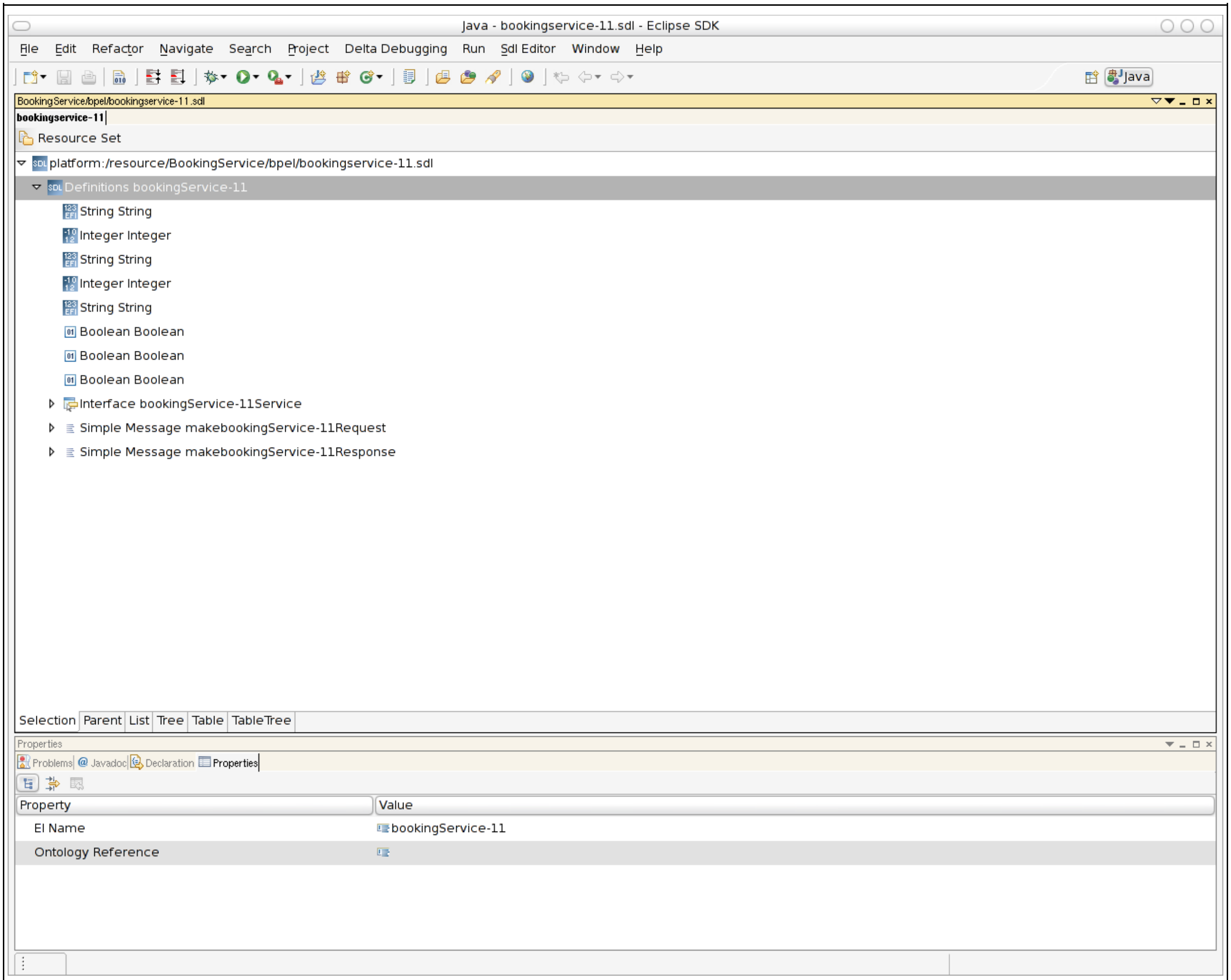


Illustration 8: SDL Editor

3.4.1 SDL Editing and Generation

As stated earlier in this document service compositions are deployed as normal DBE services and appear to clients as single atomic services, therefore these composed service need a similar interface type to atomic services. SDL is used to describe the abstract platform independent interface to a DBE service. Illustration 8 depicts a sample view of the SDL editor.
In addition to the SDL editor, users may also use the Manual Composer tool to generate a SDL interface for a given BPEL workflow with both the graphical editor and composition wizard. The task palette on the process view allows a user to generate a SDL file for their composed service. The generation uses the information provided within the BPEL definition to construct the elements and attributes of the SDL file. Although given that a BPEL definition describes services composition at an abstract level, i.e. it does not refer to any concrete implementation specifics and only references SDL interfaces, limited information is available for this generation. The composition wizard provides a more intelligent approach to SDL generation using message type matching and in process assignments, to gather the necessary parts for the input and output message types of the external SDL interface. This will be explained in more detail in sections 3.5.

3.4.2 PDD (Partner Deployment Descriptor) Editing and Generation

During the initial stage of this work-package an open source workflow engine was adopted to provide a system for executing service compositions. This engine was adapted (see section 4.1 for a more detailed description) but in the implementation of the engine it was required that a PDD

(Partner Deployment Descriptor) file was provided during the deployment of a BPEL workflow. This file tells the engine about various deployment information, e.g. references to declared PartnerLinks and to SDL files of included services (which are not explicitly declared in the BPEL description). Within the PDD file, the WS-Addressing elements EndpointReference and Address are used to describe the endpoint of the candidate DBE services. In the DBE endpoints are referenced with their respective Service Manifest ID (SMID). Therefore a PDD editor was created, in a similar fashion to the previously mentioned editors in the Manual Composer tool set. Similarly to the generation of SDL, the Manual Composer tool provides the means to generate a PDD file with both the graphical editor and composition wizard. This generation is based on the available information from a BPEL definition, in the case when using the editor, or on the selected information when using the composition wizard.

3.5 BPEL Composition Wizard

The composition wizard is used to create a new BPEL file by either starting and opening with a new BPEL process, from which a the graphical editor can be used to construct the workflow, or by continuing with a few simple wizard pages, where the user can select their services and some basic rules for the composition. It is intended that these wizard pages will help a non-BPEL developer to build a simplistic composition without requiring in-depth training of the BPEL language and all of its attributes. The user is presented with a selection of wizard pages where they can discover and select services, specify required file generation and message type matching, and select their desired workflow structure using a rule based system with the services previously selected.

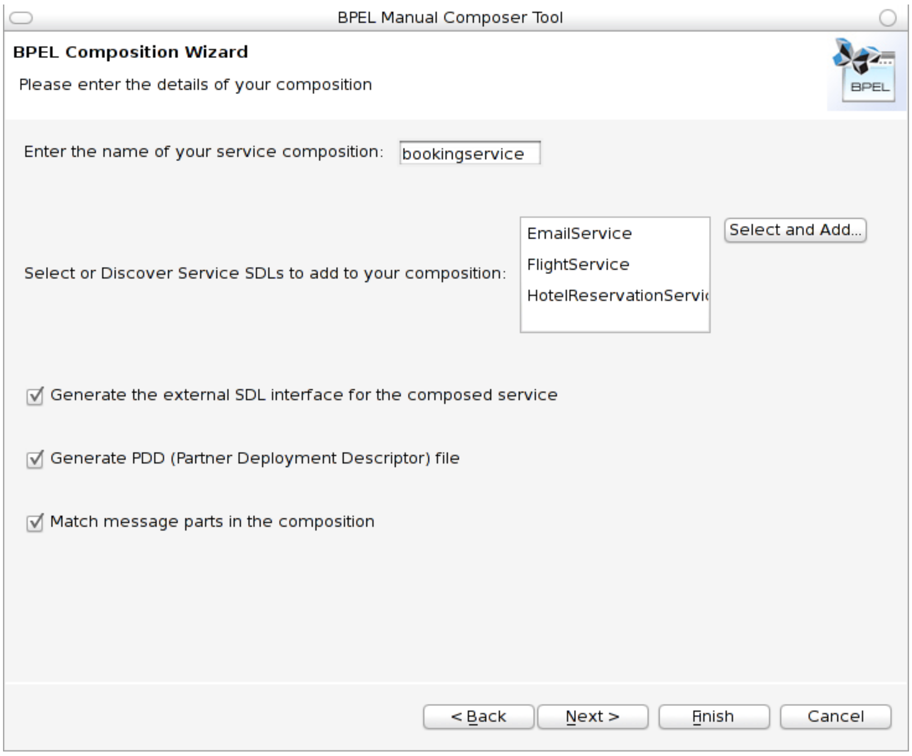


Illustration 9: Composition Wizard

3.5.1 Service Discovery and Selection

In order to create a service composition model, the user will need to discover or select previously discovered DBE services for inclusion within their desired workflow process. This discovery process can either be done in relation to the semantic descriptions of a service or the interface definition of a service. Searching algorithms and query formulations are outside the scope of this task, and are already implemented by the Query Formulator and Semantic Discovery Tool [36]. Both the composition wizard and the graphical editor have been integrated with the Query Formulator and Semantic Discovery Tool, providing a straight forward mechanism to discover

services for inclusion within a composition. When services are selected, a SMID (Service Manifest ID) is provided by the Query Formulator and Semantic Discovery Tool. This SMID is used by the wizard (or editor) to retrieve the relevant SDL definition from the Semantic Registry service. An XMI model is returned and saved to a file within the local workspace project. A local file is required during the deployment process of a BPEL composition, as the executing workflow engine requires the SDL model of each service it attempts to invoke. A URL reference can also be provided for a SDL model, which can be downloaded at run-time. In addition, local SDL files can be used during the construction of future compositions by using the 'Select and Add' button (see Illustration 10).

3.5.2 Workflow Structure Selection

Once the services are discovered, all the SDL models are parsed and their interface and operation names are listed in a series of drop down lists making up a set of rules. These rules describe a basic structure of invocations on services in sequence or parallel to each other. The limited information supplied by the user's selections in the drop down lists can enable the wizard to construct a BPEL workflow based on this information. The wizard will at some default elements and attributes to the composition which can be modified, if necessary, with the graphical editor. These elements include a default Receive activity for client requests and a Reply activity for returning information back to the client. Other default elements can include PartnerLinks, Variables and Assigns. When all the required items are selected and the rules are complete, the 'Finish' button is final pressed. This will initiate a model builder component to gather the necessary information from the wizard pages, and begin to construct the BPEL model structure.

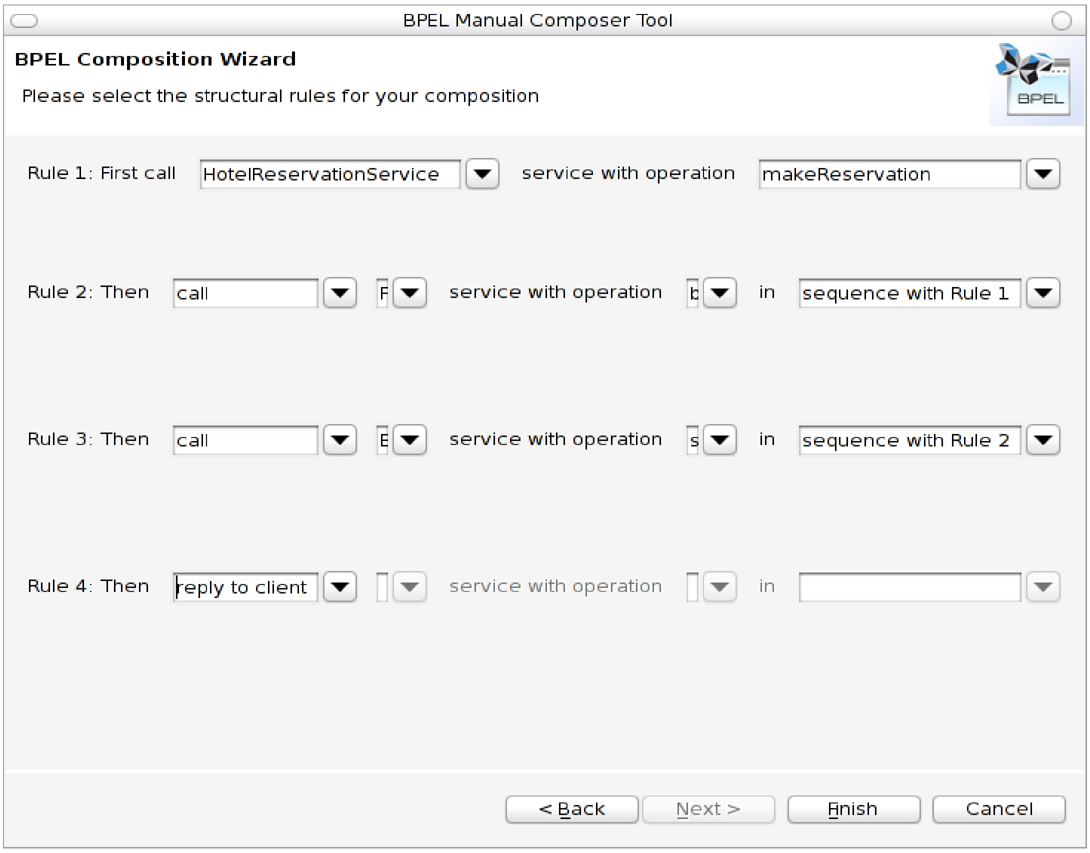


Illustration 10: Workflow Structure Selection

3.5.3 Message Type Matching

During the construction of the BPEL model it is necessary to create some connection between the inputs and outputs of all the operations invoked within the composition. A message type matching option is provided on the service selection page the wizard set. Once this option is activated, the

model builder will attempt to match the message types of outputs with inputs. If inputs are not matched to an output, then this information is required, so the necessary message type parts are added to an operation in the external SDL interface of the composition. Variables are created to store the values of the message type parts and Assign activities are include where information needs to be copied between variables. This matching process is optional due to the fact that the sometimes obvious connection between inputs and outputs of the same message type may not be the desired design of the model. All variables and assignments may be modified or removed with the graphical editor. Accurate message type matching is necessary for automatically creating executable compositions, but is very difficult to ensure the correct matching of inputs and outputs based purely on syntactical definitions. Therefore, the generated variable and assignment structure is provided as a suggestion for a particular composition model which may need some further user interaction to complete the matching.

3.5.4 File Generation

As discussed earlier, both the wizard and editor provide a mechanism to generate a SDL file as the external interface for the composition and a PDD file used in the deployment process. The editor can only generate incomplete files for both SDL and PDD due to the limited information available from the abstract elements within a BPEL model. However, the wizard has all the necessary information available from the required selections on the wizard pages. Using the message matching option in conjunction with the generation options for SDL and PDD files, the wizard can generate complete files. The generated SDL model provides a single interface and operation which contains a message type of the collated parts necessary to supply information to the included services. The generated PDD file includes PartnerLink information for each service and also creates the structure for SDL references and service IDs. Similarly, with the generation of variables and assignments within the BPEL model, the generated elements and attributes within the newly created SDL and PDD files, may need some user interaction, via the SDL and PDD editors, to ensure that the complete composition is ready for execution by clients.

3.6 BPEL Example: Purchase Order Process

The following example is taking from the BPEL4WS 1.1 specification [3] and was implemented using the BPEL editor from the Manual Composer tool. The process follows a common business process example where a business receives a purchase order request for a certain product, which then needs to be processed and an invoice to be sent back to the customer. Once the purchase order is received from a customer, the process is initiated. The process structure mainly follows the parallel execution of three tasks: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order. While some of the processing can proceed concurrently, there are control and data dependencies between the three tasks. In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three tasks are completed, invoice processing can proceed and the invoice is sent to the customer, see Illustration 11. Appendix B shows the full BPEL model for this example.

The process defined in BPEL declares a set of partner links for specifying the relationships between all parties involved in the process. It defines four partner links, one for the sender of the purchase order, i.e. the customer, and one for each of the three services used within the process, i.e. the invoicing service, the shipping service and the scheduling service. The process also declares a set of variables which enable the process to maintain state and history of message exchanges. All the variables are of message types which are used during invoking, receiving and reply activities. Finally before we discuss the structure of the workflow process, it declares a fault handlers section which contains the activities which must be executed in response to faults occurring during the process. This fault handlers section contains a catch activity which if triggered will execute a reply activity, sending a message back to the client with a fault message type.

The structure of the process starts with a sequence activity, which states that all child activities

within this activity will be executed sequentially. The first basic activity, as usual, is a receive activity. When a client sends a purchase order request for this process, the receive activity is triggered which in turn activates the process instance. This is because the createInstance attribute is set to “yes”. The port type, operation and message type (as a variable) are equivalent to those declared in the external WSDL definition of the composed service.
The process continues after the initial receive activity with a flow structural activity. Within this flow activity, two link elements are declared which will be used to synchronize the concurrent activities to be executed. The flow activity declares three activities, all sequence activities, which are to be executed in parallel. The first sequence activity is intended to invoke the shipping service. Initially the customer’s information, which was received earlier, is copied to a variable used in the invocation of the service. An operation is invoked on the shipping service to request shipping relating to the customers request. A source element is declared here using the link, ship-to-income, which will be explained later. Finally, the last activity in this initial parallel task, is a receive activity. This is used to allow for asynchronous messaging to service, where the previously invoked service, i.e. the shipping service, will use the call-back mechanism to return the result to the process. This activity waits for the shipping schedule and also declares a source element using the link, ship-to-scheduling.

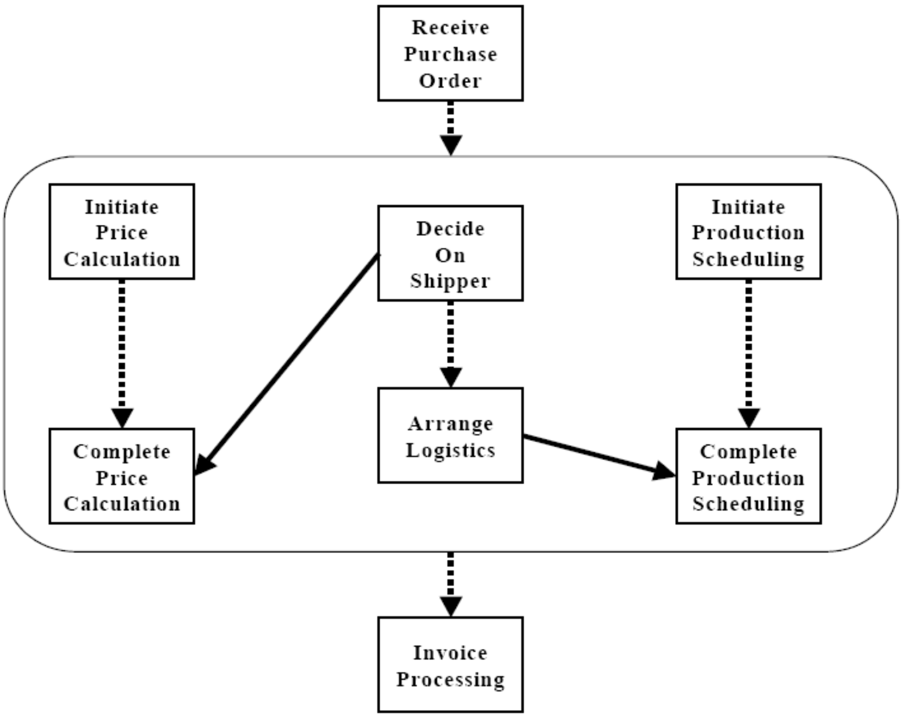


Illustration 11: Purchase Order Process Flow Diagram

The second task to be executed concurrently is declared by two invoke activities followed by a single receive. The first activity invokes an operation on the invoicing service to initiate the price calculation, while the second invocation on a different operation of the same service requests the shipping price to be sent back. The second invoke activity cannot be completed until the previous invocation of the shipping service is finished and the shipping information is returned form the shipping service, as declared by the target element using the link, ship-to-invoice. This task is completed with a receive activity which waits for an asynchronous call-back from the invoicing service with the invoice data.
The final task to be executed is declared by two invoke activities within the sequence structure. The first invoke executes an operation on the scheduling service to request the scheduling of production for the customer’s initial request. The second invoke also sends a message to the scheduling service

but to an operation which sends the shipping schedule. This activity must wait for the shipping service to call back to the process with the shipping schedule before it can be executed, as declared by the target element using the link, ship-to-scheduling. Finally, if all tasks have been performed correctly by their respective activities and no fault was thrown then the flow activity is finished. One activity remains in the process to return the invoice data to the customer. This reply activity synchronously responses to the initial request accepted to the first receive activity.

4 Manual Composition and Runtime Infrastructure

Manual Composition describes how a workflow process is created and how the atomic services are combined within a service composition model. Once this model has been designed, it needs to be deployed and made available so that it can be executed within the run-time environment of the DBE. As an additional effort within the Manual Composer task, we decided to adopt an open source BPEL workflow engine. The obvious choice was to use the ActiveBPEL engine, as it was a better and more complete implementation than the other open source alternatives, called Twister, at the time of research.

4.1 Workflow Engine

During the early stage of working with the ActiveBPEL workflow engine, it was deemed necessary to make some changes to this open source engine. Firstly, to be integrated within the DBE Execution Environment, extensions were made to the engine to provide the mechanism for Fada [41] lookups and dynamic invocations at run-time. The modifications involved providing a custom DBE invoke handler for the ActiveBPEL engine. This invoke handler extracts the SMID from the WS-Addressing scheme in the PDD file and downloads a service proxy from FADA and invokes on the generic interface of that proxy.

The original ActiveBPEL workflow engine was implemented to run within the Tomcat web container as a servlet application. This application was integrated, but loosely coupled, with the Axis Web Services container to provide the entry point of SOAP clients and the communication point to other web services during the execution of a BPEL workflow. See Illustration 12 for a flowchart of a web service request. The execution of a BPEL workflow process starts within the engine when one of its start activities is triggered, either by an incoming message or by a Pick activity's alarm. Each BPEL process must have at least one start activity. The engine dispatches incoming messages to the correct process instance, but if there is correlation data, the engine tries to find the correct instance that matches the correlation data, otherwise the request matches a start activity and a new process instance is created [7].

When a BPEL process model is deployed to the engine, the definition will be inspected and the engine will create objects, called activity definitions, which model the process. Both the engine and its event listeners have access to these definitions. The events contain an XPath value, defined in the activity definitions, that indicates which activity in the process is triggering the event. By using a Visitor pattern, the engine visits the activity definition object model and creates the implementation objects. The engine encapsulates any implementation logic within this construction process, e.g. an implicit scope within an invoke activity will generate an explicit scope with a single invoke child activity. The designer, or even other listeners, can remain ignorant to these implementation decisions since they're only aware of the activity definitions and their XPath information relating from the BPEL model. [7].

Currently executing receive activities from all process instances are stored and queued in the receive queue. These receive activities include onMessage activities that are part of a pick or an event handler. A receive activity is said to be executing when it has been queued by its parent activity but has not yet received the message that it's waiting for from the outside world. For example, during the execution of a structural activity, such as a sequence activity, a receive activity is next to be executed. In this case, the sequence activity would block and wait for a message to be received therefore at it to the receive queue. The receive queue also contains inbound messages from the other partners or clients that did not match up to a waiting receive activity already in the queue and were themselves not capable of creating a new process instances as the createInstance attribute of the receive activity was not set to "yes". An unmatched message like this is possible given the asynchronous nature of some Web services and business processes. The engine will accept these unmatched messages provided that they contain correlation data, but the messages are only queued until a timeout period passes [7].

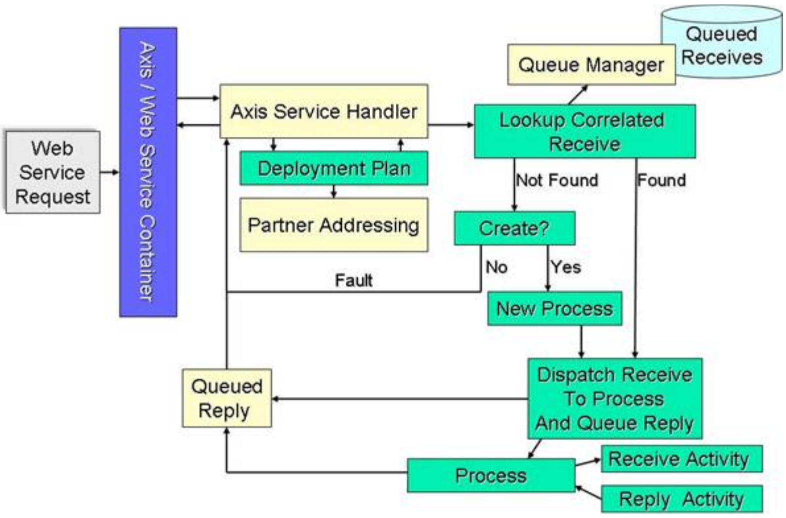


Illustration 12: Request Dispatch Flow Diagram

If a process blocks and waits at an activity, like a receive, then this activity will be queued until the data arrives or the process terminates. Pick activities are slightly different to receive activities. The first onMessage or onAlarm to match for the pick immediately sets the state of all of the other possible messages/alarms for that pick to DEAD_PATH state (see the table below for all activity states) and therefore removing them from the receive queue. Event handlers automatically remove their queue entries once the scope that defines them completes. Each activity has an associated and these states change based on the meta-model of BPEL. The activities also fire events to notify listeners of their changes in state and there are mechanisms in place for listening to these events [7]. An activity must be in one of the following states:

Activity States	*Description*
INACTIVE	All BPEL activities are in the inactive state when the Process starts
READY_TO_EXECUTE	Ready to execute. These activities have been queued by their parent and their join condition has evaluated to true
EXECUTING	Currently executing
FINISHED	Finished executing without a fault
FAULTED	Finished executing with a fault
DEAD_PATH	Removed from the execution path due to dead path elimination. When a parent activity's state becomes DEAD_PATH, that state is propagated to all of its children
QUEUED_BY_PARENT	Queued for execution by their parents
TERMINATED	Terminated
Unknown	The activity's state is null. If a parent activity's state becomes unknown, then the children's' states change to INACTIVE.

4.2 Dynamic Discovery and Invocation of Services

The first modification of the workflow engine was to introduce the mechanism for dynamic discovery of service proxies at run-time during the execution of any BPEL process. With the original implementation of the engine, the locations of services were generally discovered at design time contained within the binding information of a service's WSDL definition. Although, it was not essential that a WSDL definition of each service used in the process was present at the deployment of the workflow. A designer could specify static or dynamic endpoint references, but there is no

external mechanism for discovering the endpoint or even the services at run-time. Following the approach taken in the design of the DBE Execution Environment, the engine would discover service instances using Fada [41]. To discover these service instances the engine needs a reference to the registered service, i.e. the SMID (Service Manifest ID) of the service. Once the engine had knowledge of this SMID, then during an attempted invocation of a service, the engine would perform a lookup on a Fada node, using the Proxy Framework, with the SMID as a Fada entry. If a registered service instance was found corresponding to the SMID, then the Service Proxy object of this service would be downloaded to the engine. The Service Proxy object contains the location or endpoint reference to the actual remote service. The implementation of this involved extending a set of handlers which were primarily used for making Axis SOAP calls directly to the pre-located service. To make the SMID available to the engine, it was added at design time to the partner deployment descriptor file which accompanies each BPEL model during deployment.

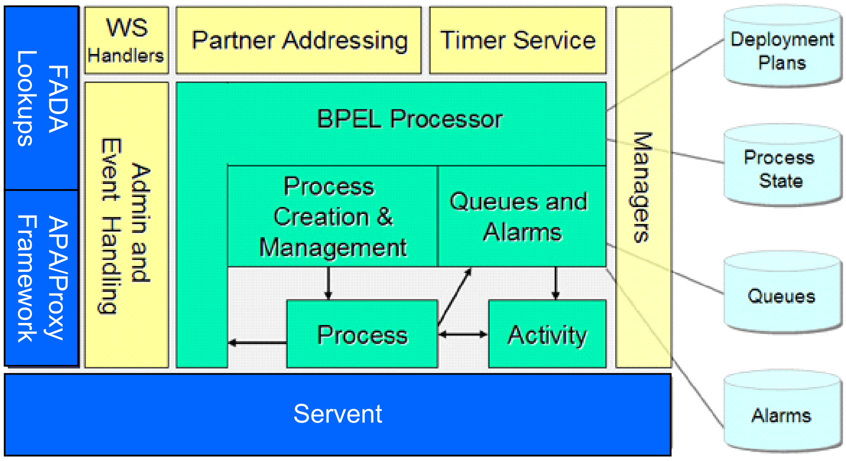


Illustration 13: Adapted BPEL Workflow Engine

In addition to containing a reference to the location of a service, the Service Proxy object downloaded after a successful Fada lookup also contains the necessary structure to perform a dynamic invocation of a service with a variety of communication protocols. The structure provided by the Proxy Framework allows certain properties to be encapsulated into a Service Proxy object. These properties can specify protocol types, endpoint addresses, service names, UI factories, and other extensible features necessary for service usage. Along with the structure of the Proxy Framework, another framework called the APA (Abstract Protocol Adapter) was integrated at this stage. The APA provides a common interface for generically invoking an operation on any service while making the invocation independent of the protocol used. The APA can be extended to implement any number of protocol adapters which can create the necessary setup for a dynamic invocation of a remote service. Such protocol adapters have been made for SOAP, kSOAP, and object serialisation over HTTP.

4.3 Platform and Container Independence

The modelling of service interface definitions within the DBE has been profiled as PIM (Platform Independent modelling) as specified by the MDA. The interface of each service is described by SDL and incorporated into the SM of each service. Therefore it would make sense to use these abstract interface definitions within the deployment and execution of workflow services. Of course a difficultly arose with the fact that the BPEL specification has close links with WSDL and the Web Services architecture. WSDL provides an abstract interface definition of a service but it also provides concrete binding information which tells clients how to bind and locate services using a specific platform approach, i.e. Web Services.
Substantial work has begun on designing an effective approach to integrating the execution of BPEL models with the platform independent SDL models. Fortunately, with the lose coupling of the workflow engine and the Axis Web Services container, the development work is minimal

enough to justify the continued adoption of the ActiveBPEL engine. The Axis dependency has been replaced with the Proxy Framework and APA, as stated previously in section 4.2, for the invocation of services. Some issues still exist with the differences in the abstract part of WSDL's and SDL's meta-models. WSDL defines a 'PortType' element where as SDL defines an 'Interface'. The BPEL meta-model does specify a PortType attribute which relates to the PortType of a given WSDL service. This PortType attribute can be substituted by the Interface name in the SDL definition.
In addition to the previously described modifications to the workflow engine, it has been designed to remove the dependency on a servlet container, such as Tomcat, so that the workflow engine can be more easily integrated within the Servent architecture. Already, the workflow engine has been deployed on a more light-weight servlet container, called Jetty, and integrated into the Servent application.

5 Towards Autonomous Service Composition

Dynamic discovery of service proxies and dynamic invocation of services is an initial step towards the composition of services in a semi-automated or autonomous way. For another task in the Composer work-package, called autonomous composition (formally called the Automatic Composer), which will begin during the second phase of the project, research has been done already for so as to design the initial Manual Composer tool with the future goal in mind. It is assumed that the work between both tasks, manual composition and autonomous composition, will be interlinked as the project progresses. Therefore the users and developers of the DBE will have a set of integrated intelligent tools for creation of dynamic and static adaptable service compositions. From our research we have realised some terminology which may help to clarify the extent of the tasks being undertaking.

An autonomous service composition is, by definition, an autonomous process, where an agent or some application actively manages and adapts a deployed composition. Autonomous composition can enable a service composition to be independent and self-directing with regards to the services it interacts with or even the order of these interactions, i.e., no human interaction will be necessary for such actions to be taken. This may require that the workflow process maintains some intelligence, whether initially declared by the creators or gained through a variety of run-time relationships. These relationships will affect the decision making of the process of execution within the workflow. Autonomous compositions offer the possibility of:

- Self-repairing faults in compositions, e.g., if a service in a composition is unavailable then the service can be replaced with an alternative working version
- Self-optimising compositions, e.g., if a service in a composition is performing poorly then the service can be replaced with an alternative working version

Manual composition refers some sort of manual or user interaction either at design-time and/or run-time; where autonomous composition suggests some partial or semi-automatic composition. Semi-automatic composition may describe the initial manual composition of a static workflow with dynamic features at design-time, but with the addition of suggestive data, possibly policy based information. An intelligent workflow engine can make autonomous decisions regarding the direction of the dynamic features within the static workflow. These dynamic features could be service selection prior to invocation, conditional structure flow within the workflow and perhaps the scheduling of activities. Some of these dynamic features, like service selection, may have different approaches, e.g. if the discovery and selection relates to a specific interface then this process is relatively straight forward, although if the discovery and selection process relates to semantic descriptions then a process of service interface matching will be needed to distinguish whether the interface of a service is compatible with the workflow definition. The selection of services may also involve other structural services within the DBE, such as the Recommender Service. The Recommender Service [38] may suggest candidate services relating to the interface or semantic attributes required by the static workflow. Initial work has already been implemented for semi-automated composition within the DBE, with the use of dynamic discovery of service proxies and dynamic invocation of services within the adapted ActiveBPEL workflow engine.

Automatic composition does not necessarily require a static workflow. Here, the workflow structure and attributes can be created and modified automatically at run-time even without any manual intervention. Although in the more common scenario, there will be a need for an initial manually composed static workflow, but this workflow can be adapted during execution by intelligent agents guided by policies or rules to achieve the optimal process flow for a given service composition. Again this will also involve the dynamic features of semi-automatic composition, like service selection at run-time. Most of these concepts are at an early stage within the research community and there also are many issues which could arise from business and contractual conflicts of automatic service selection. Therefore, this research expands into the areas of automated contract

agreements, ad-hoc business partner relationships and even automated licensing consolidation.
This brief chapter has outlined some research concepts and approaches to autonomous composition. The characteristics described here are not considered as requirements for the task of autonomous composition. Autonomous composition can been seen as a viable approach towards more automated service composition, although some of the advanced features discussed in this chapter may be too early for the current evolution of SOAs and out of the scope of this work-package. Further work is necessary to decide on the feasibility of these approaches and the integration with the tools described in this document. The Autonomous Composition task will explore this field further and provides a first implementation to address more dynamic service compositions. This will be part of the deliverable D17.4 Autonomous Service Composition Engine.

6 Conclusion

Service composition can be viewed as creating a generic pluggable process, where services are atomic objects within an organised chain, or as a complex workflow process specifying strict interactions and activities between well defined services. A pluggable approach offers ease of use to users but reduces the power and complexity of the resultant service composition. A comprehensive modelling approach enables developers to create complex adaptive workflows but requires specialised knowledge. Depending on which view is taken, a set of tools are required to design, deploy and execute these compositions. This document described the current implementations and design for ongoing aims to provide the users of the DBE with intelligent and intuitive tools for the emerging possibilities of service composition.

As outlined in this document, service composition is a large but relatively emergent research area, with limited standards and open source implementations available. Many approaches were investigated and it was decided upon to use a widely-used composition language and an existing open source workflow engine. With the adoption of these and other technologies, it enabled the creation of the initial foundation for service composition within the DBE. The tools developed and integrated allow for much more than building simple service compositions by complex workflow models and dynamic executions. Manual composition is just an initial stage in the extensive area of service composition with the DBE. The ongoing and future work under the Manual Composer task is been directed towards the creation of a more adaptive, flexible and powerful tools and infrastructure. Substantial research has been done towards the other interlinked tasks within this work-package.

The existing foundation for service composition already implemented for the DBE has set a premise for the further development of advanced intelligent tools which will reach into all areas of current research. Such sophisticated tools will assist in the evolution of the DBE to serve users with competitive and progressive advantages within a dynamic SME environment.

7 Glossary

Term	*Description*
BML	Business Modelling Language
BPEL	Business Process Execution Language: A workflow orchestration language which defines a process-centric model for the formal specification of the behaviour of business processes based on the interaction of the process and it partners.
EMF	Eclipse Modelling Framework: A Java modelling framework for building tools and other applications based on a structured data model
GEF	Graphical Editing Framework: An Eclipse framework which allows developers to easily create a rich graphical editor providing representations for existing model.
KB	Knowledge Base: Is the part of the DBE system where the DBE knowledge is stored and managed. Such Knowledge refers to ontologies, business and service.
KB Service	Knowledge Base Service: A service on top of the DBE KB that provides functionality for storing and retrieving models.
MDA	Modern Driven Architecture: An approach (proposed by OMG) to IT system specification that separates the specification of the system functionality for the specification of the implementation of that functionality on a specific technology.
OWL-S	Ontology Web Language for Web Services: Web services are enhanced with semantic descriptions in ontologies that are computer-interpretable, which is an important precondition for automatic discovery, composition and execution of Web services
PDD	Partner Deployment Descriptor: A deployment descriptor file specific to deploying BPEL workflows on the ActiveBPEL engine.
Recommender	A DBE (autonomous) core service that will provide users with personalised knowledge by exploiting their profiles.
SOA	Service-Oriented Architecture: A component model that inter-relates the different functional units of an application, called services, through well-defined interfaces and contracts between these services.
SDL	Service Definition Language: A MOF model (meta-model) that provides technical definition of the programmatic interface of a service.
SM	Service Manifest: This represents the service when it is associated with a supplier or vendor, containing business specific data (the BML M0 level). It is published and hence it is an offered service, it has public visibility and is available to consumers.
WS-CDL	Web Services Choreography Description Language: A language used to model multi-party collaborations which describe the externally observable behavior of peer services and their clients by specifying the message exchanges between them.
WSDL	Web Service Definition Language

8 References

[1] Chris Peltz. Web Service Orchestration – a Review of Emerging Technologies, Tools, and Standards. Hewlett Packard. 2003

[2] Biplav Srivastava, Jana Koehler. Web Service Composition – Current Solutions and Open Problems. IBM India and Zurich Research Laboratory. International Conference on Automated Planning and Scheduling, 2003

[3] Microsoft, IBM, Siebel Systems, BEA, SAP. Business Process Execution Language for Web Service, version 1.1. May 2003. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[4] David Martin, Mark Burstein, Grit Denker. OWL-S 1.0 Release. http://www.daml.org/services/owl-s/1.0/

[5] The Large Scale Distributed Information Systems Lab. METEOR-S: Semantic Web Services and Processes. http://lsdis.cs.uga.edu/Projects/METEOR-S/

[6] ActiveBPEL, LCC. The ActiveBPEL Open Source BPEL Engine. http://www.activebpel.org/index.html

[7] ActiveBPEL, LCC. The ActiveBPEL Engine Architecture. http://www.activebpel.org/docs/architecture.html

[8] Twister Sourceforge.net Project. Twister WS-BPEL Workflow Engine. http://www.smartcomps.org/twister/index.html

[9] The Eclipse Foundation (Borland, IBM, ..). The Eclipse Open Extensible IDE. http://www.eclipse.org

[10] The Eclipse Foundation (Borland, IBM, ..). Eclipse Modelling Framework. http://www.eclipse.org/emf

[11] The Eclipse Foundation (Borland, IBM, ..). Graphical Editing Framework. http://www.eclipse.org/gef

[12] OWL Services Coalition. OWL-S: Semantic Markup for Web Services. November 2003. http://www.daml.org/services/owl-s/1.0/owl-s.html

[13] Boualem Benatallah, Quan Z. Sheng, Marlon Dumas. The Self-Serv Environment for Web Services Composition. IEEE Internet Computing Jan/Feb 2003 Vol 7 No 1

[14] Anis Charfi, Mira Mezini. Aspect-Oriented Web Service Composition with AO4BPEL. European Conference on Web Services (ECOWS) 2004

[15] Daniel J. Mandell, Sheila A. McIlraith. A Bottom-Up Approach to Automating Web Service Discovery. Customization, and Semantic Translation. Proceedings of the Second International Semantic Web Conference (ISWC 2003)

[16] R. Aggarwal, K. Verma, J. Miller and W. Milnor. Constraint Driven Web Service Composition in METEOR-S. IEEE International Conference on Services Computing (SCC) 2004

[17] P. Rajasekaran, John Miller, Kunal Verma, Amith Shet. Enhancing Web Services Description and Discovery to Facilitate Orchestration. Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC) 2004, Part of the 2nd International Conference on Web Services (ICWS) 2004

[18] Jorge Cardoso, Amit Sheth. Introduction to Semantic Web Services and Web Process Composition. First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)

[19] UDDI, Universal Description, Discovery and Integration. www.uddi.org

[20] Matjaz Juric. BPEL and Java. An Article from TheServerSide.com. April 2005

[21] Mohamad Afshar, Hal Hilderbrad, Nickolaos Kavantzas. Process-centric realization of SOA: BPEL moves into the limelight. Web Services Journal. November 2004

[22]Antoine Lonjon. Challenges and Methods for the Implementation of Service Oriented Architecture. BP-Trends. April 2005

[23]OASIS. Web Services Business Process Execution Language Technical Committee. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[24]IBM Developer Works. New to SOA and Web Services. http://www-128.ibm.com/developerworks/web-services/newto/

[25]Object Management Group (OMG). Corba Basics. http://www.omg.org/gettingstarted/corbafaq.htm

[26]Object Management Group (OMG). http://www.omg.org

[27]Organization for the Advancement of Structured Information Standards. www.oasis-open.org

[28]World Wide Web Consortium, W3C. www.w3.org

[29]Web Services Interoperability Organization, WS-I. www.ws-i.org

[30]Microsoft. COM: Component Object Model Technologies. http://www.microsoft.com/com/

[31]Michael N. Huns, Munindar P. Singh. Service-Oriented Computing: Key Concepts and Principles. IEEE, Internet Computing, Jan/Feb 2005 Ed.

[32]Web Services Choreography Working Group. www.w3.org/2002/ws/chor/

[33]Alistair Barros, Marton Dumas, Phillipa Oaks, A Critical Overview of the Web Services Choreography Description Language (WS-CDL). BPTrends March 2005

[34]Steve Ross-Tablot. Dancing with Web Services: W3C chair talks choreography. SearchWebServices.-com. March 2005

[35]IBM Alphaworks. Business Process Execution Langauge for Web Services Java Runtime (BPWS4J). http://www.alphaworks.ibm.com/tech/bpws4j

[36]Trinity College Dublin, Del 24.1: DBE First Implementation. 2005

[37] Soluta.Net, Del 21.2: Architecture Scoping Document. 2004

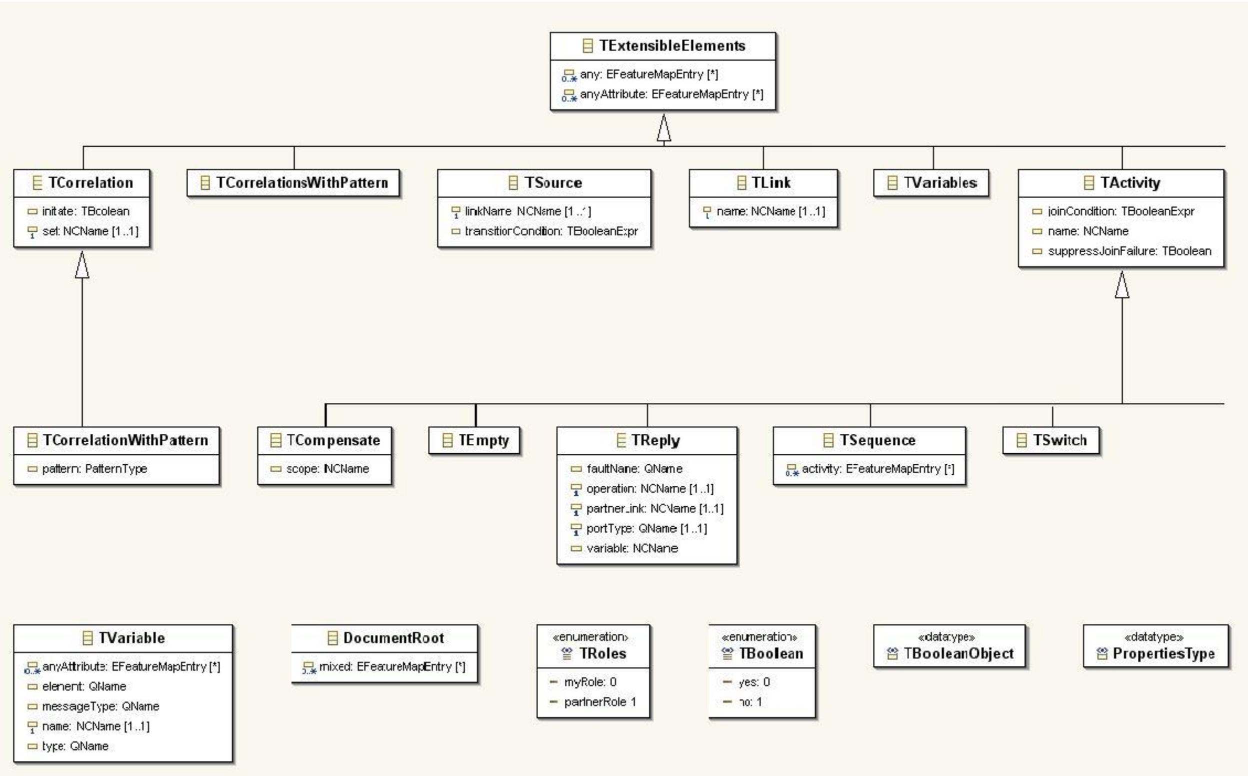
[38]Technical University of Crete, Del 17.1: Recommender. 2005

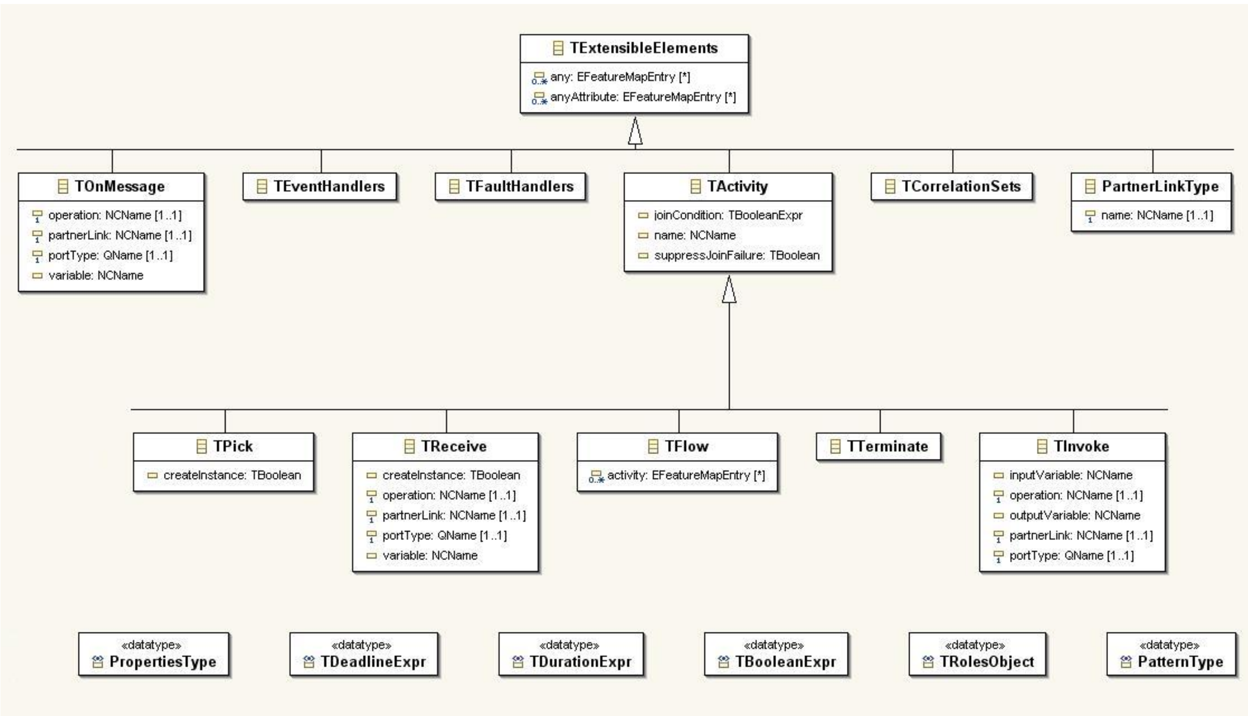
[39]W3C. Web Services Choreography Description Language, version 1.0 (working draft). December 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/

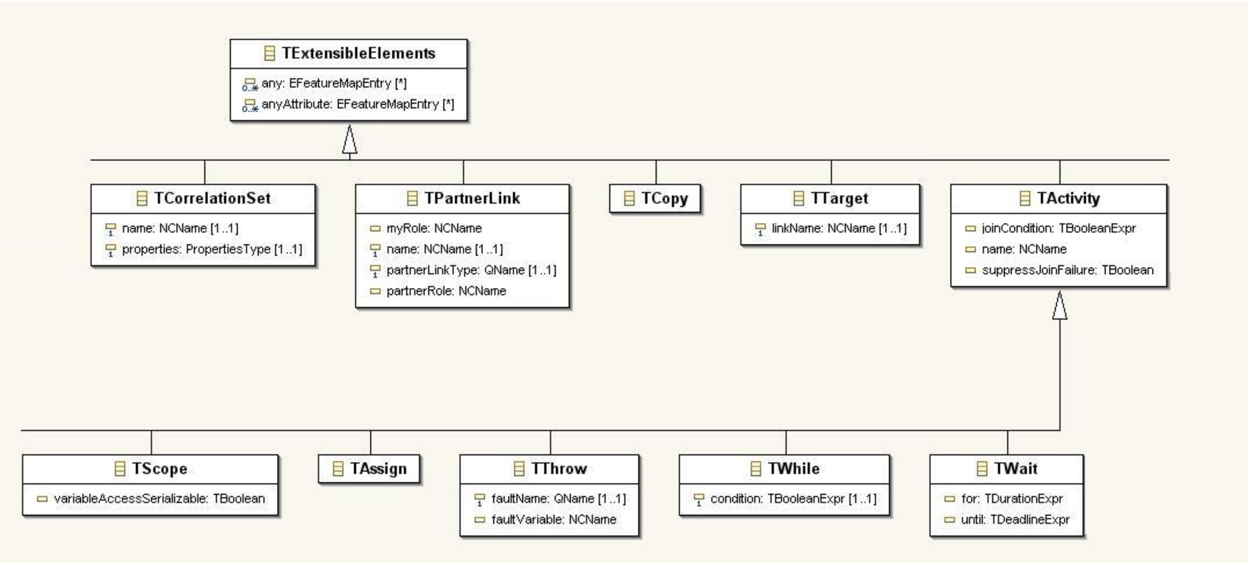
[40]Soluta.Net, Del 16.1: Service Definition Language. 2005

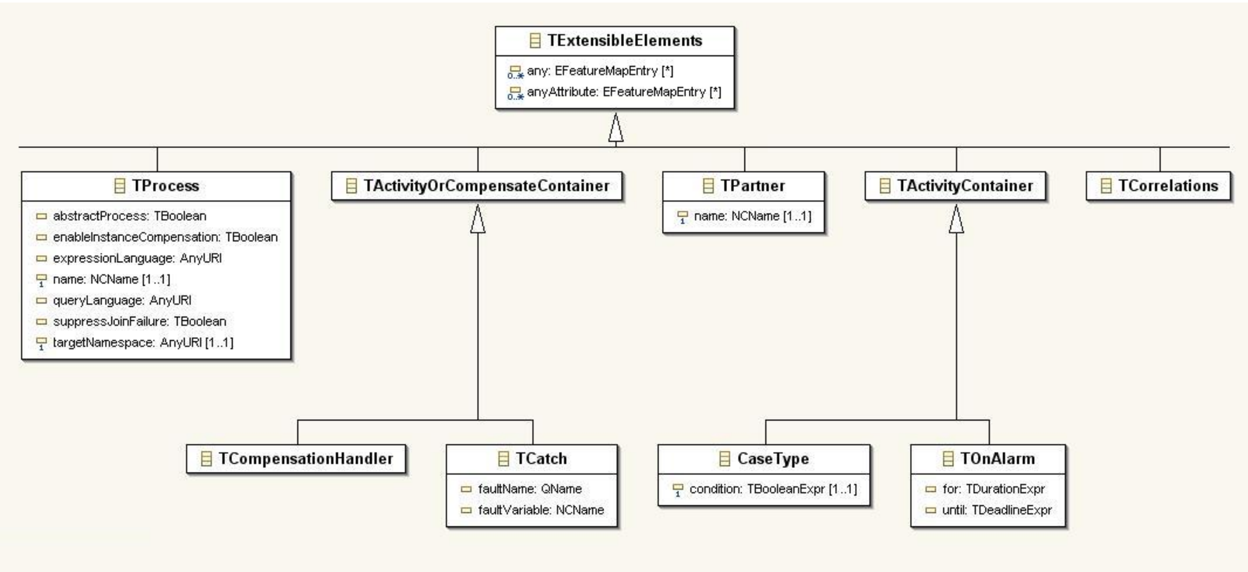
[41]Techideas. FADA Overview and Core FADA. http://fada.techideas.info/index.html

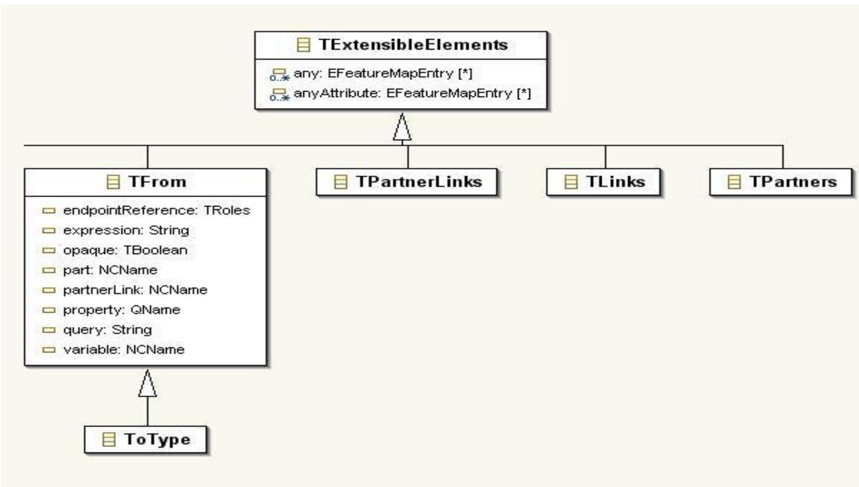
9 Appendix A: BPEL Metamodel






TExtensibleElements
any: EFeatureMapEntry [*]
anyAttribute: EFeatureMapEntry [*]
TCorrelationSet
name: NCName [1..1]
properties: PropertiesType [1..1]
TPartnerLink
myRole: NCName
name: NCName [1..1]
partnerLinkType: QName [1..1]
partnerRole: NCName
TCopy
TTarget
linkName: NCName [1..1]
TActivity
joinCondition: TBooleanExpr
name: NCName
suppressJoinFailure: TBoolean
TScope
variableAccessSerializable: TBoolean
TAssign
TThrow
faultName: QName [1..1]
faultVariable: NCName
TWhile
condition: TBooleanExpr [1..1]
TWait
for: TDurationExpr
until: TDeadlineExpr


TExtensibleElements
any: EFeatureMapEntry [*]
anyAttribute: EFeatureMapEntry [*]
TProcess
abstractProcess: TBoolean
enableInstanceCompensation: TBoolean
expressionLanguage: AnyURI
name: NCName [1..1]
queryLanguage: AnyURI
suppressJoinFailure: TBoolean
targetNamespace: AnyURI [1..1]
TActivityOrCompensateContainer
TPartner
name: NCName [1..1]
TActivityContainer
TCorrelations
TCompensationHandler
TCatch
faultName: QName
faultVariable: NCName
CaseType
condition: TBooleanExpr [1..1]
TOnAlarm
for: TDurationExpr
until: TDeadlineExpr


TExtensibleElements
any: EFeatureMapEntry [*]
anyAttribute: EFeatureMapEntry [*]
TFrom
endpointReference: TRoles
expression: String
opaque: TBoolean
part: NCName
partnerLink: NCName
property: QName
query: String
variable: NCName
TPartnerLinks
TLinks
TPartners
ToType

10 Appendix B: BPEL Example Model – Purchase Order Process

```
<process name="purchaseOrderProcess"
         targetNamespace="http://acme.com/ws-bp/purchase"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:lns="http://manufacturing.org/wsdl/purchase">
  <partnerLinks>
    <partnerLink name="purchasing"
                 partnerLinkType="lns:purchasingLT"
                 myRole="purchaseService"/>
    <partnerLink name="invoicing"
                 partnerLinkType="lns:invoicingLT"
                 myRole="invoiceRequester"
                 partnerRole="invoiceService"/>
    <partnerLink name="shipping"
                 partnerLinkType="lns:shippingLT"
                 myRole="shippingRequester"
                 partnerRole="shippingService"/>
    <partnerLink name="scheduling"
                 partnerLinkType="lns:schedulingLT"
                 partnerRole="schedulingService"/>
  </partnerLinks>

  <variables>
    <variable name="PO" messageType="lns:POMessage"/>
    <variable name="Invoice" messageType="lns:InvMessage"/>
    <variable name="POFault" messageType="lns:orderFaultType"/>
    <variable name="shippingRequest" messageType="lns:shippingRequestMessage"/>
    <variable name="shippingInfo" messageType="lns:shippingInfoMessage"/>
    <variable name="shippingSchedule" messageType="lns:scheduleMessage"/>
  </variables>

  <faultHandlers>
    <catch faultName="lns:cannotCompleteOrder"faultVariable="POFault">
      <reply partnerLink="purchasing"
             portType="lns:purchaseOrderPT"
             operation="sendPurchaseOrder"
             variable="POFault"
             faultName="cannotCompleteOrder"/>
    </catch>
  </faultHandlers>

  <sequence>
    <receive partnerLink="purchasing"
             portType="lns:purchaseOrderPT"
             operation="sendPurchaseOrder"
             variable="PO"
             createInstance="yes">
    </receive>
    <flow>
      <links>
        <link name="ship-to-invoice"/>
        <link name="ship-to-scheduling"/>
      </links>
      <sequence>
        <assign>
          <copy>
            <from variable="PO" part="customerInfo"/>
            <to variable="shippingRequest" part="customerInfo"/>
          </copy>
```

```
        </assign>
        <invoke partnerLink="shipping"
                portType="lns:shippingPT"
                operation="requestShipping"
                inputVariable="shippingRequest"
                outputVariable="shippingInfo">
          <source linkName="ship-to-invoice"/>
        </invoke>
        <receive partnerLink="shipping"
                 portType="lns:shippingCallbackPT"
                 operation="sendSchedule"
                 variable="shippingSchedule">
          <source linkName="ship-to-scheduling"/>
        </receive>
      </sequence>
      <sequence>
        <invoke partnerLink="invoicing"
                portType="lns:computePricePT"
                operation="initiatePriceCalculation"
                inputVariable="PO">
        </invoke>
        <invoke partnerLink="invoicing"
                portType="lns:computePricePT"
                operation="sendShippingPrice"
                inputVariable="shippingInfo">
          <target linkName="ship-to-invoice"/>
        </invoke>
        <receive partnerLink="invoicing"
                 portType="lns:invoiceCallbackPT"
                 operation="sendInvoice"
                 variable="Invoice"/>
      </sequence>
      <sequence>
        <invoke partnerLink="scheduling"
                portType="lns:schedulingPT"
                operation="requestProductionScheduling"
                inputVariable="PO">
        </invoke>
        <invoke partnerLink="scheduling"
                portType="lns:schedulingPT"
                operation="sendShippingSchedule"
                inputVariable="shippingSchedule">
          <target linkName="ship-to-scheduling"/>
        </invoke>
      </sequence>
    </flow>
    <reply partnerLink="purchasing"
           portType="lns:purchaseOrderPT"
           operation="sendPurchaseOrder"
           variable="Invoice"/>
  </sequence>
</process>
```

11 Appendix C: BPEL Manual Composer Tool – User Guide

The BPEL Manual Composer Tool provides a set of integrated editors and wizards from which a studio developer can compose a BPEL (Business Process Execution Language) workflow for execution within the DBE. The central component of this tool is a BPEL graphical editor, which comprises of a three views: a BPEL process overview, a graphical tree editing view, and a source editing view. In addition to editing BPEL files, a user can avail of a composition wizard to more simply construct their BPEL workflows. With the wizard, a user can select or discover services to include within the newly created workflow. The user must then specify some simple rules which will then be used to build the structure of the workflow and generate the BPEL file. In addition to this BPEL file, the wizard can be used to generate an external interface to the workflow in the format of SDL (Service Definition Language) and a partner deployment descriptor file (PDD file).
The BPEL Manual Composer Tool has been designed in conjunction with the DBE workflow engine which has been implemented to support DBE services.
The BPEL Manual Composer Tool is provided as one eclipse plugin called 'org.dbe.studio.editors.bpel_x.y.z'. This plugin is dependent on the Eclipse installation (version 3.1.x), all EMF plugins (version 2.1.x), GEF plugins (version 3.1.x), and the Studio Query Formulator and Semantic Discovery Tool, org.dbe.studio.tools.qfsdt_0.2.0.

11.1 BPEL Composition Wizard

The composition wizard provides a user with an optional means to build a BPEL workflow by using a set of simple wizard pages. The user can directly open the BPEL graphical editor or initially use the composition wizard to select inclusive SDL services, specify the structure of the workflow, provide automated matching of message types and then generate the BPEL file with the additional SDL interface and PDD file of the workflow. The BPEL graphical editor and composition wizard can be selected from within the Composer perspective of the DBE Studio: File -> New -> BPEL Graphical Editor and Composition Wizard.

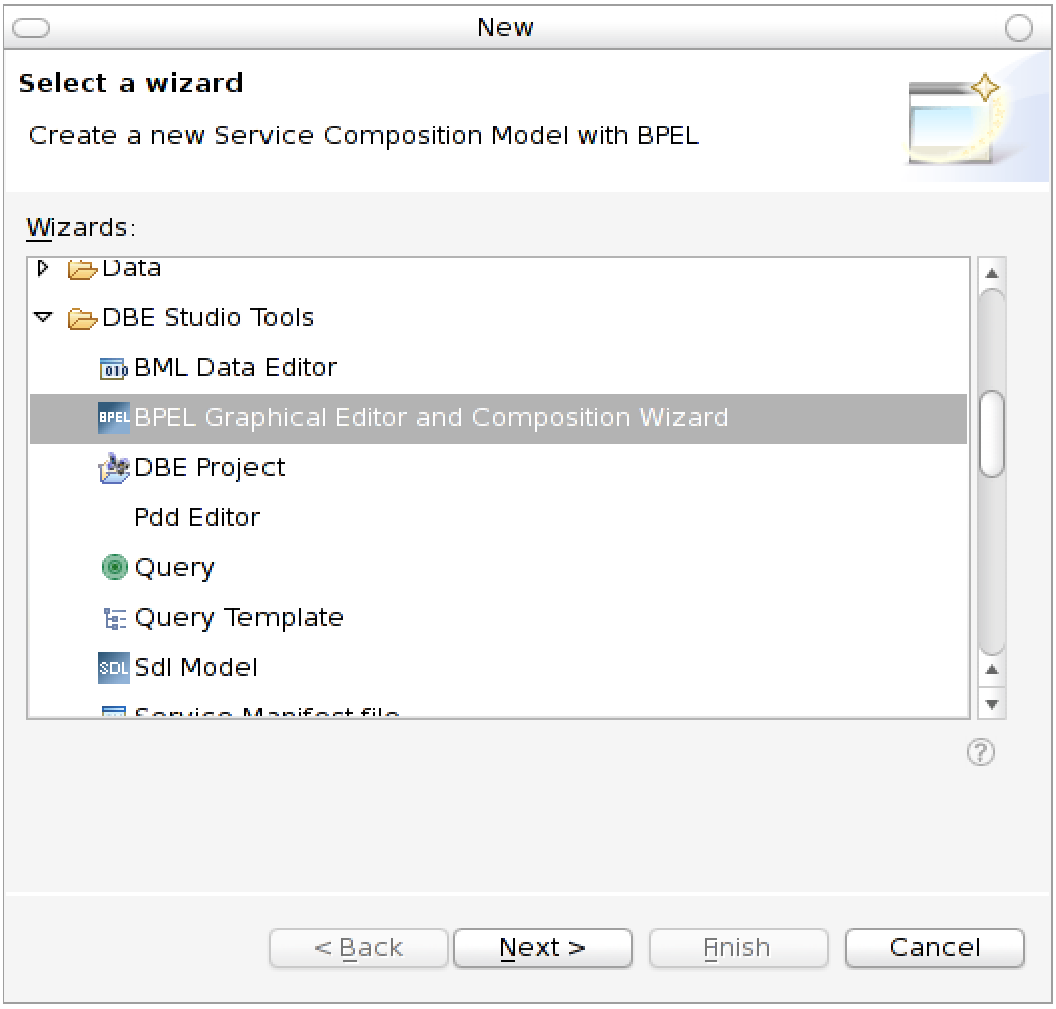


Illustration 14: Selecting the BPEL Graphical Editor and Composition Wizard

To create a BPEL file a parent directory or project must be selected and file name entered with the extension '.bpel'.

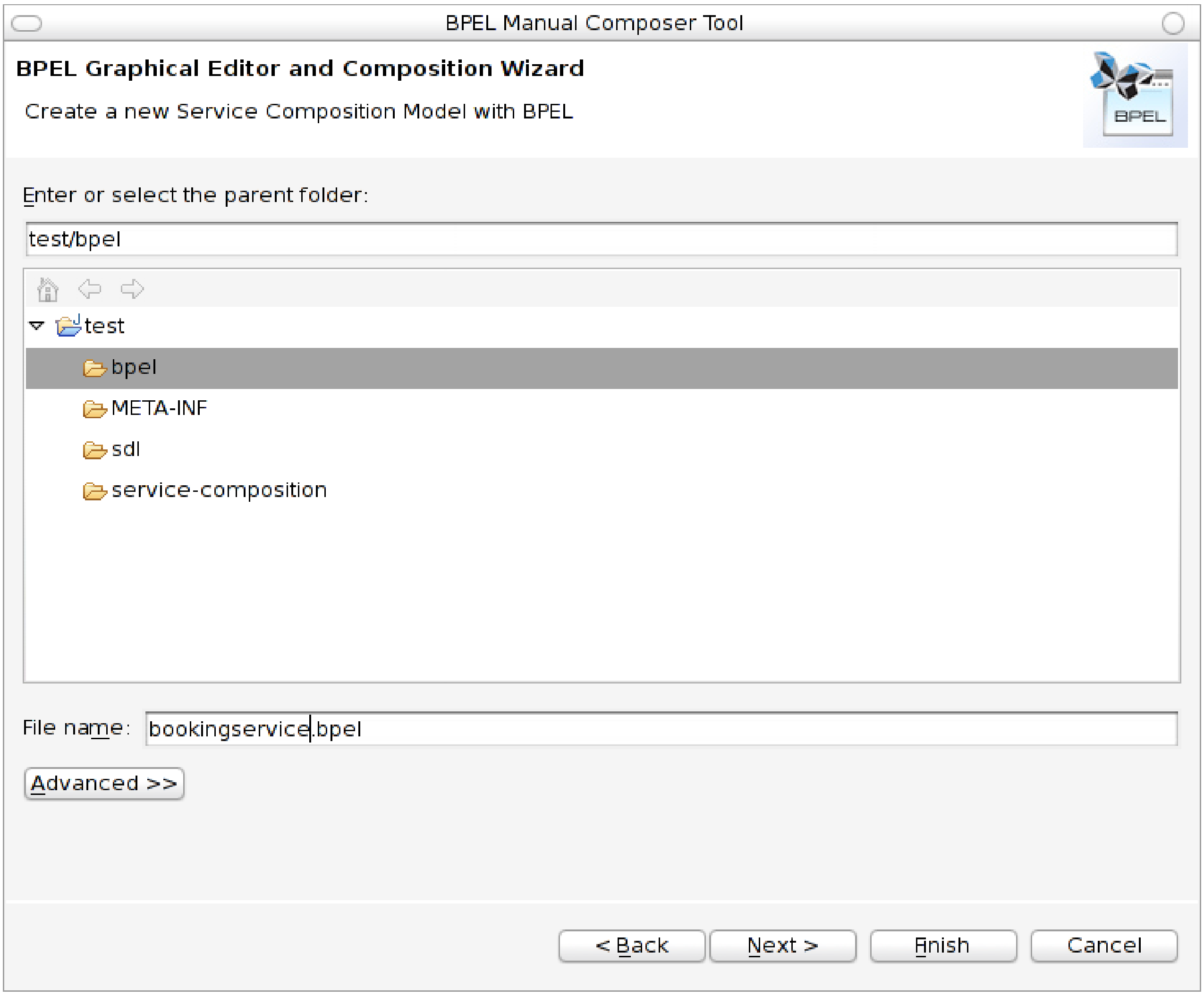


Illustration 15: Selecting Parent Folder and Entering BPEL File Name

After clicking Next, the user is presented with the option of opening the BPEL grahical editor or continuing with the BPEL composition wizard. Click Finish for the editor or click Next for the wizard. (Please note that the composition wizard will open the graphical editor when it is finished).

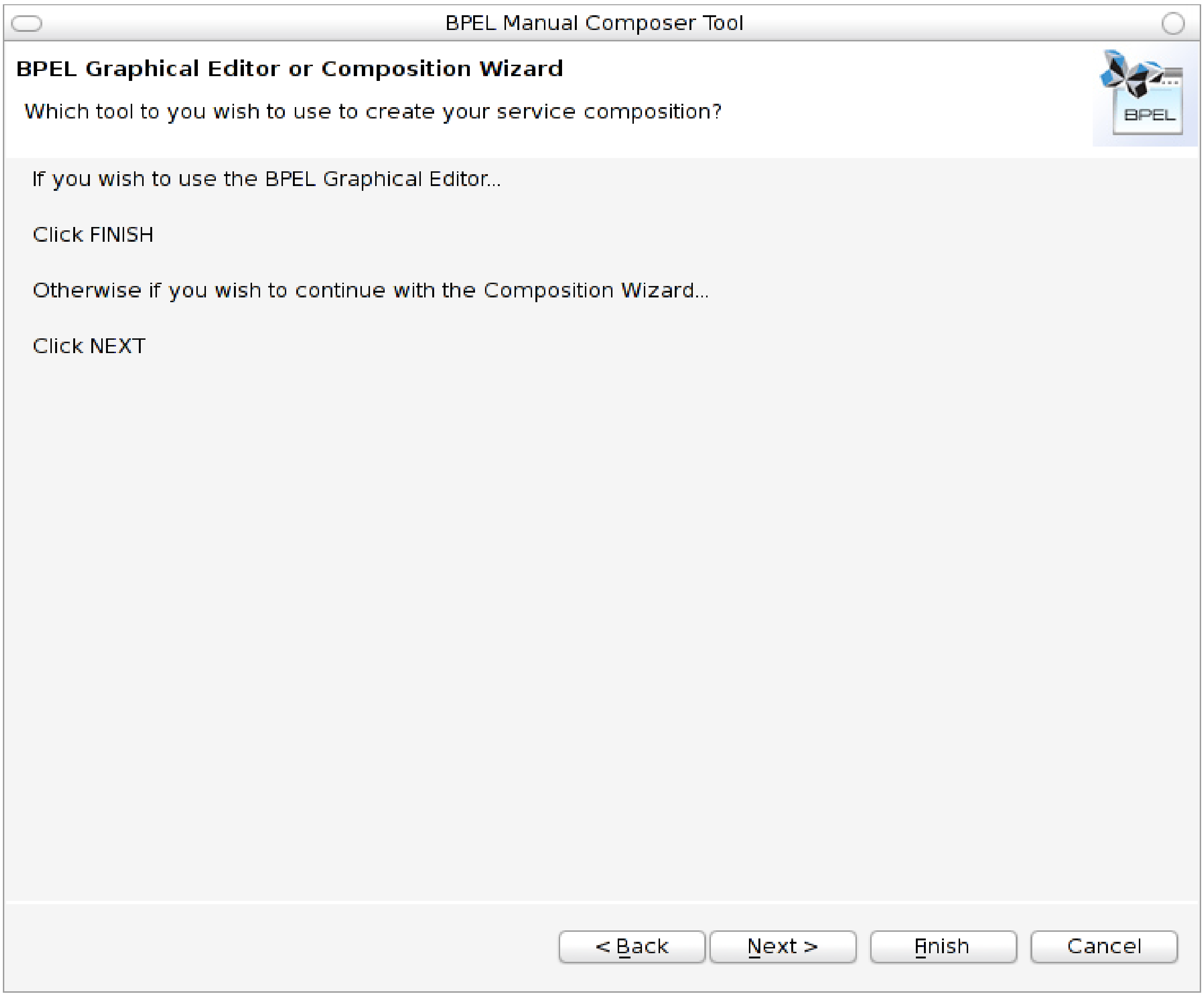


Illustration 16: Choose to Open the BPEL Graphical Editor or Continue with the Composition Wizard

After clicking Next, the first page of the composition wizard is presented to the user. This page requires the user to enter the name of the composition, select the SDL services which are to be included within the workflow, and select other generation and message part matching options. The SDL generation option will create a SDL interface for the BPEL workflow which includes the required inputs for the other services included in the composition. The PDD file generation option will create a PDD file which is required for deployment with the associated workflow engine.

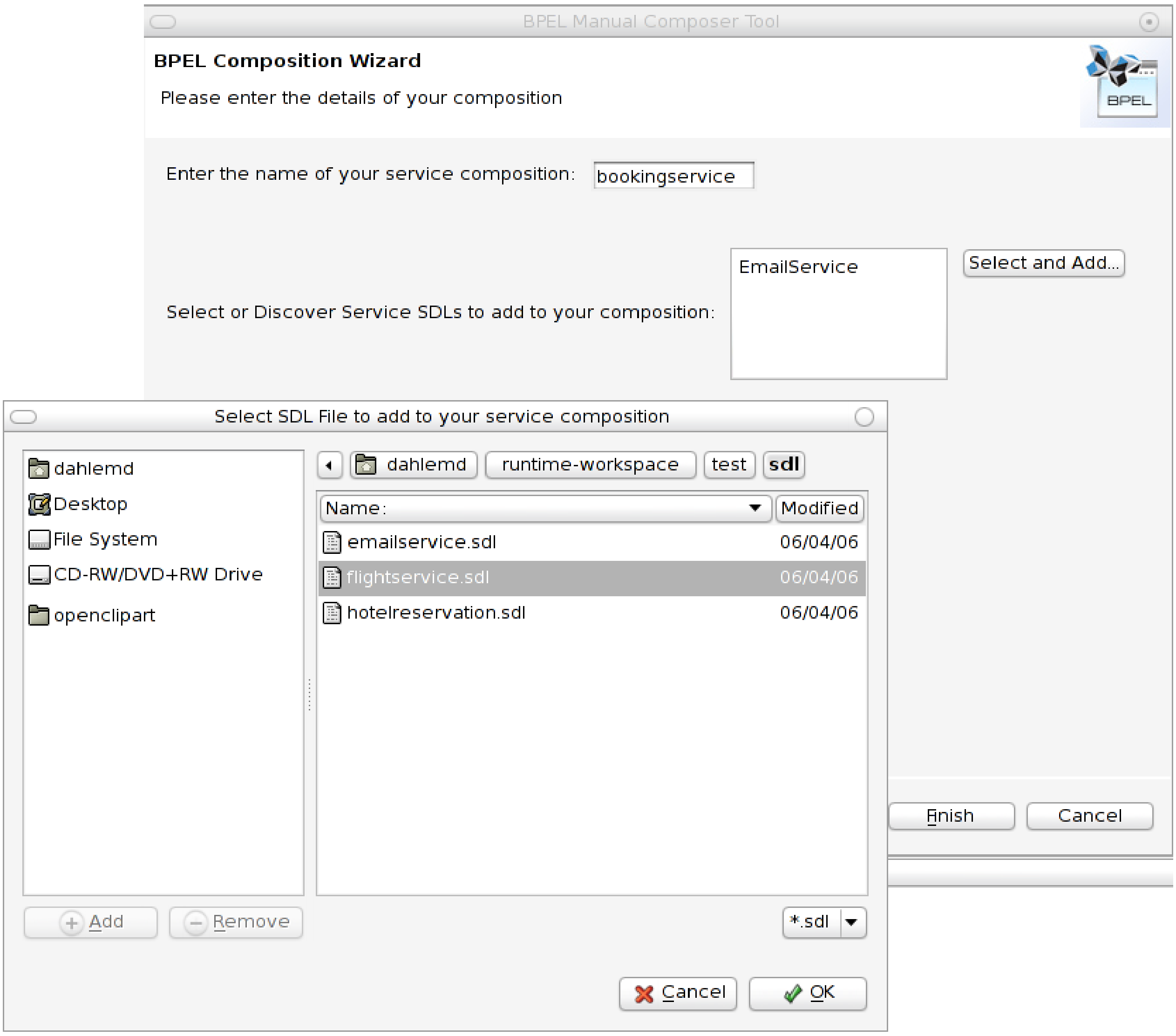


Illustration 17: BPEL Composition Wizard, Select SDL Services Page

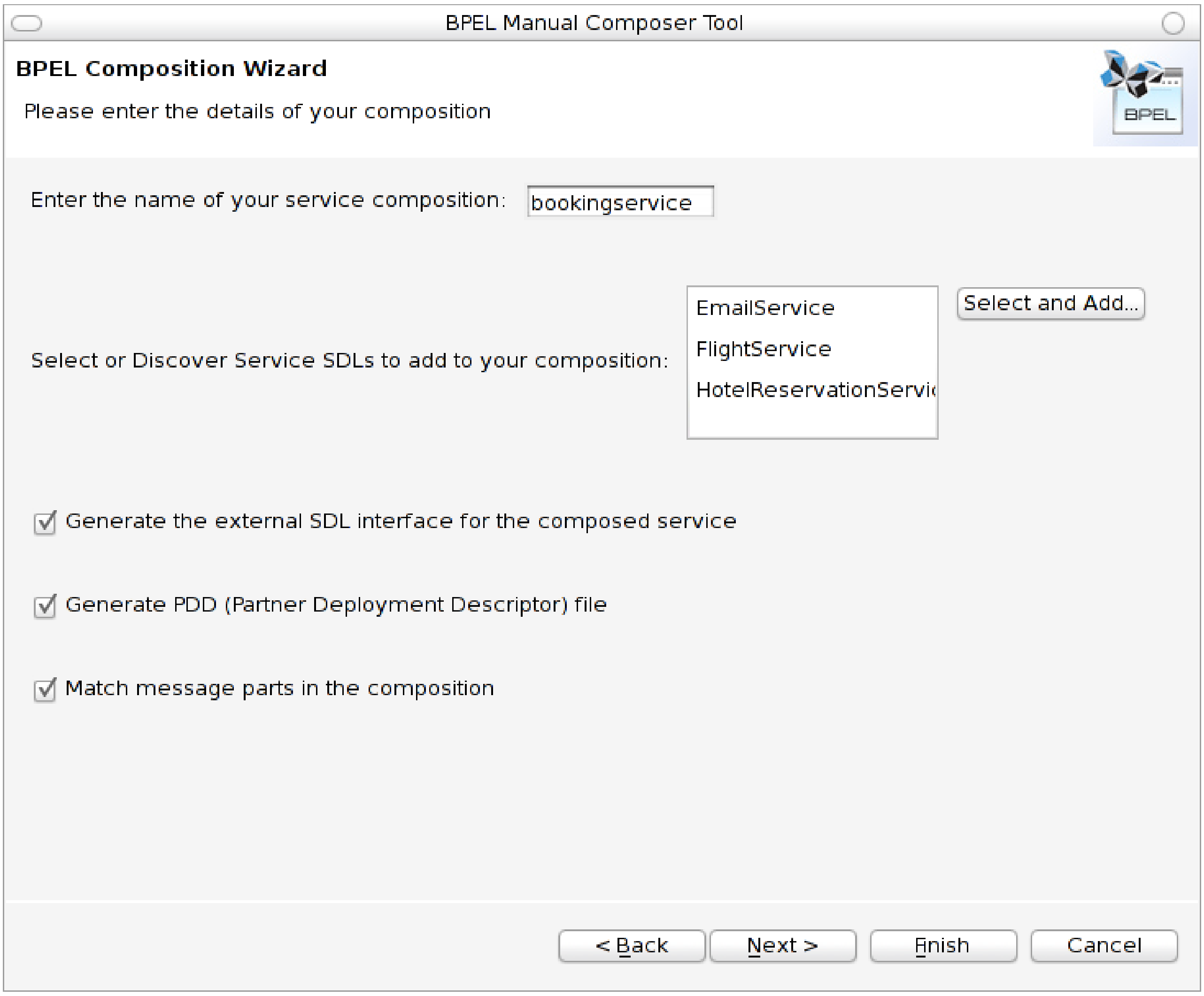


Illustration 18: Select Other Generation and Matching Options

The second page of the composition wizard provides a selection of drop down lists arranged as part of a set of structural rules for the composition. The drop down lists includes the interface names, operation names and structural activities, such as sequence or parallel activities. When the desired rules are selected and the Finish button is clicked, the required BPEL structure is generated and the BPEL graphical editor is opened.

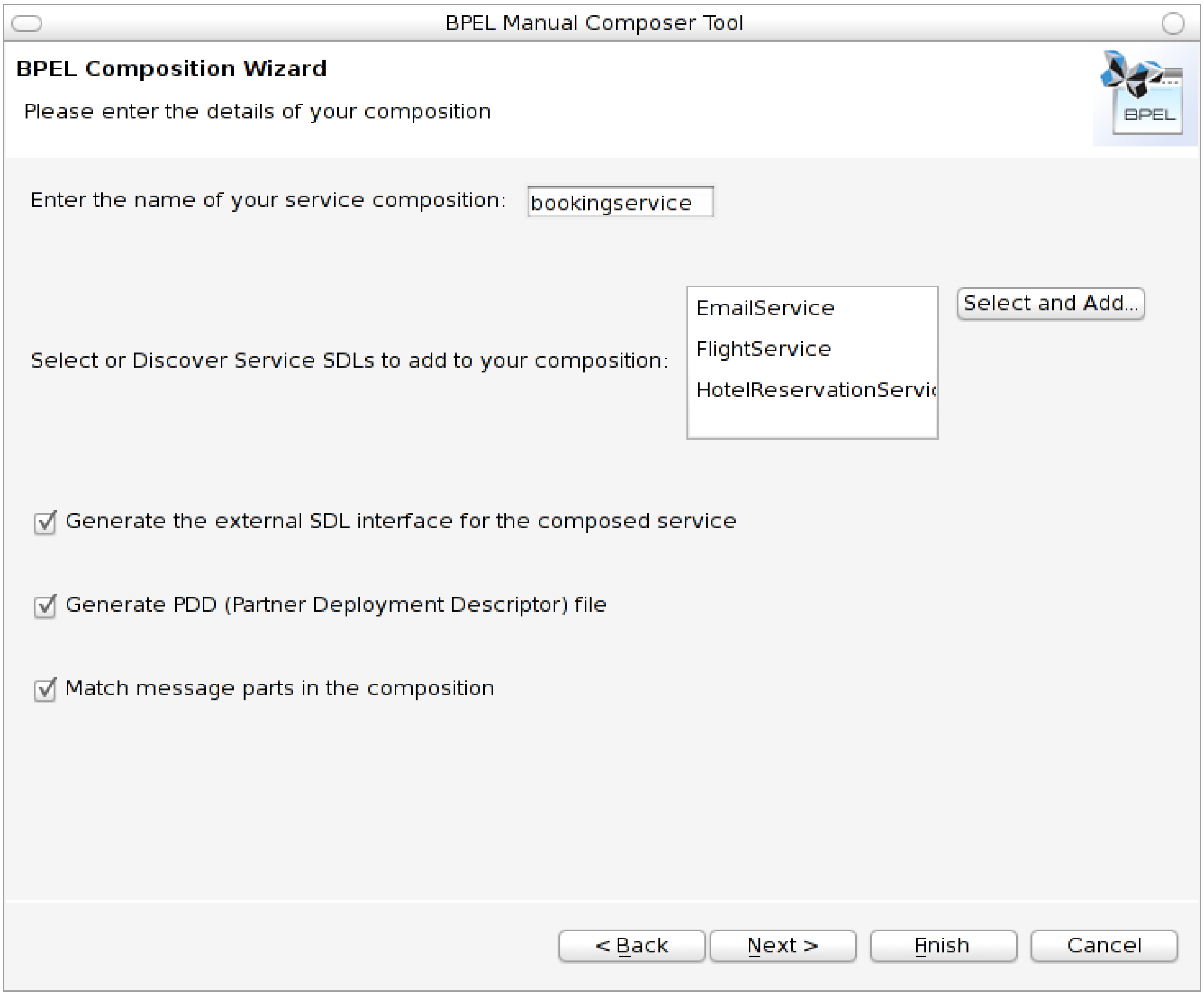


Illustration 19: Specify the Structural Rules for the Composition

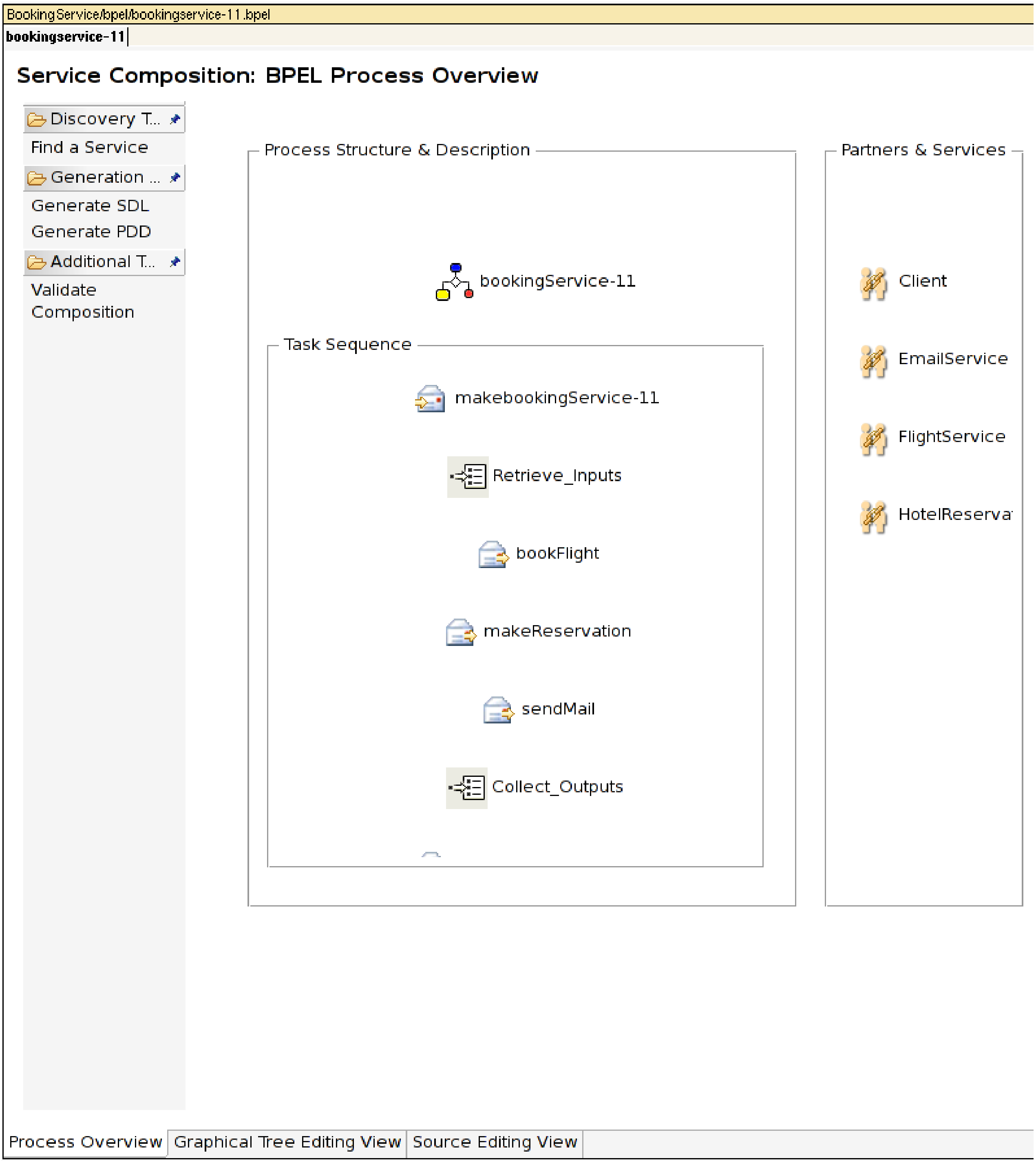


Illustration 20: BPEL Workflow created as result of using the BPEL Composition Wizard

If the generation options were selected for the above BPEL workflow, then the following SDL and PDD files would be created with the necessary parts added.

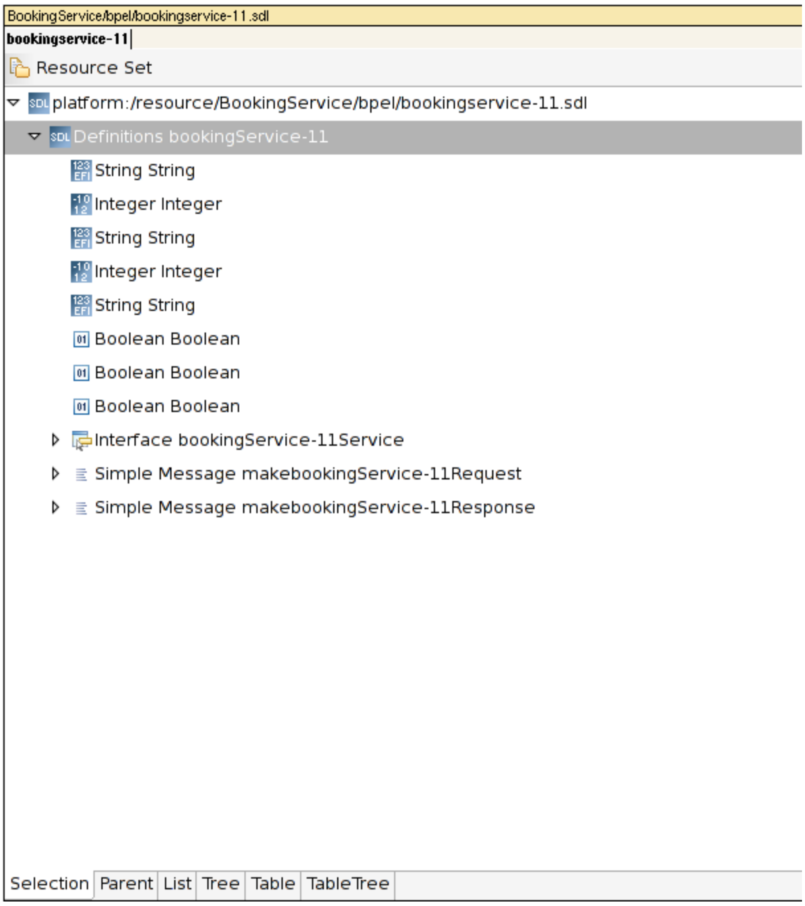


Illustration 21: Generated SDL file

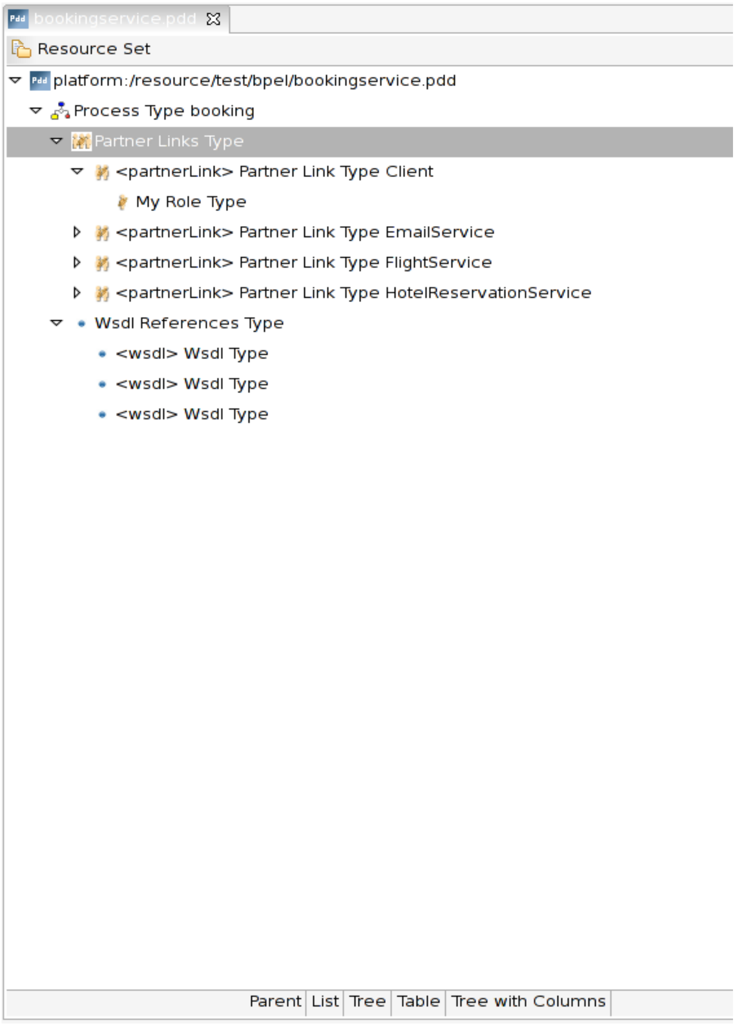


Illustration 22: Generated PDD file

11.2 BPEL Graphical Editor

The main feature of the BPEL Manual Composer Tool is a BPEL graphical editor. This editor provides three views: a BPEL process overview, a graphical tree editing view, and a source editing view. The BPEL process overview is first visible view when the editor is opened. This view provides a graphical summary of the BPEL process and its internal structure and activities. It also shows a list of the partners/services included in the composition. To the left-hand side of this view is a task palette where additional operations can be performed. These tasks include service discovery, SDL and PDD file generation and BPEL validation. The individual tasks can be accessed by clicking on the task button and then clicking anywhere in the 'Process Structure & Description' box.

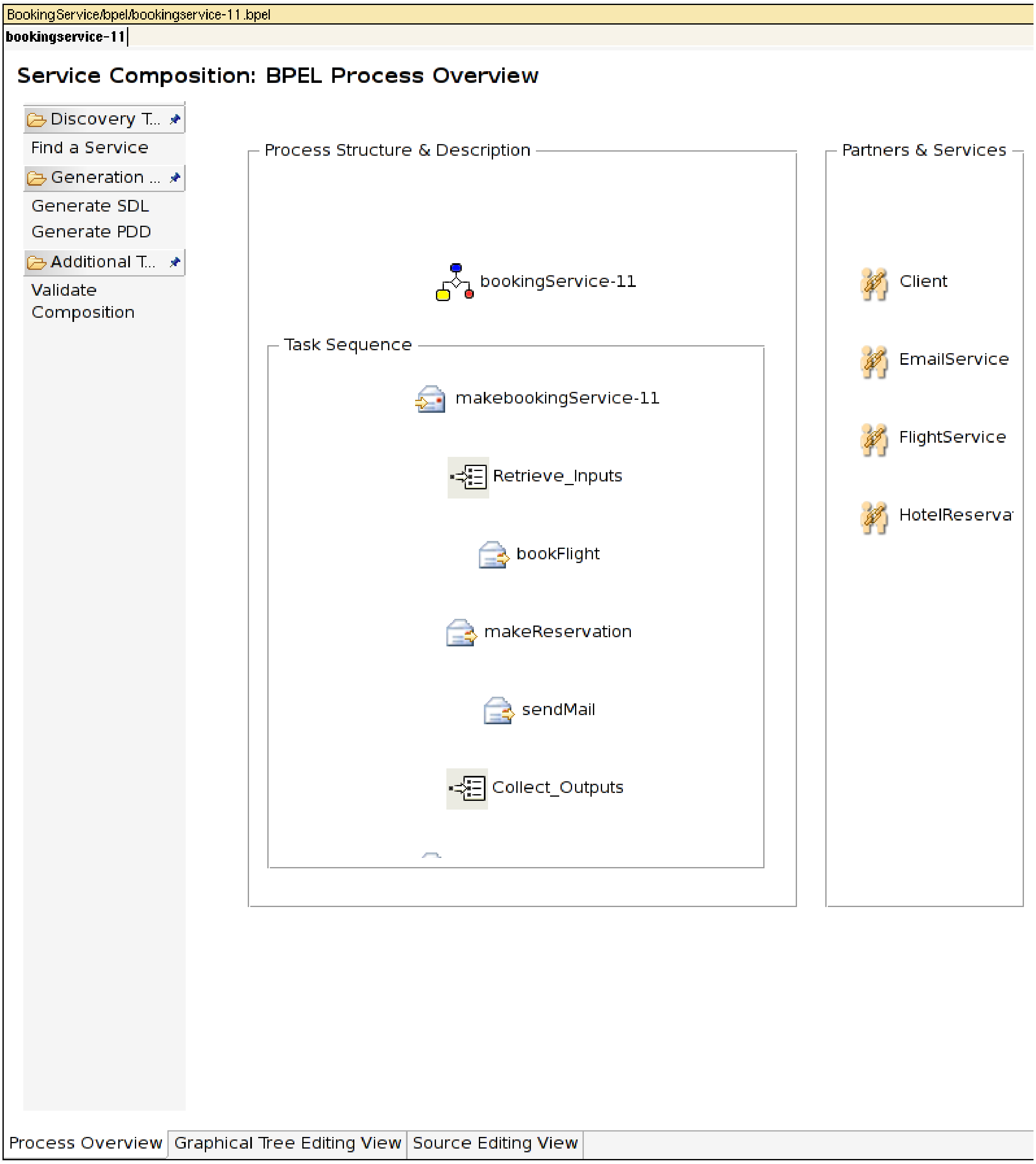


Illustration 23: BPEL Editor: Process Overview

The graphical tree editing view provides a tree based editor for constructing and modifying BPEL structures and activities. The BPEL process is the root element within any BPEL workflow. All activities within a process have associated properties which can be accessed and modified by a properties view. If this view is not visible, right click anywhere in the editor and click 'Show Properties View'.

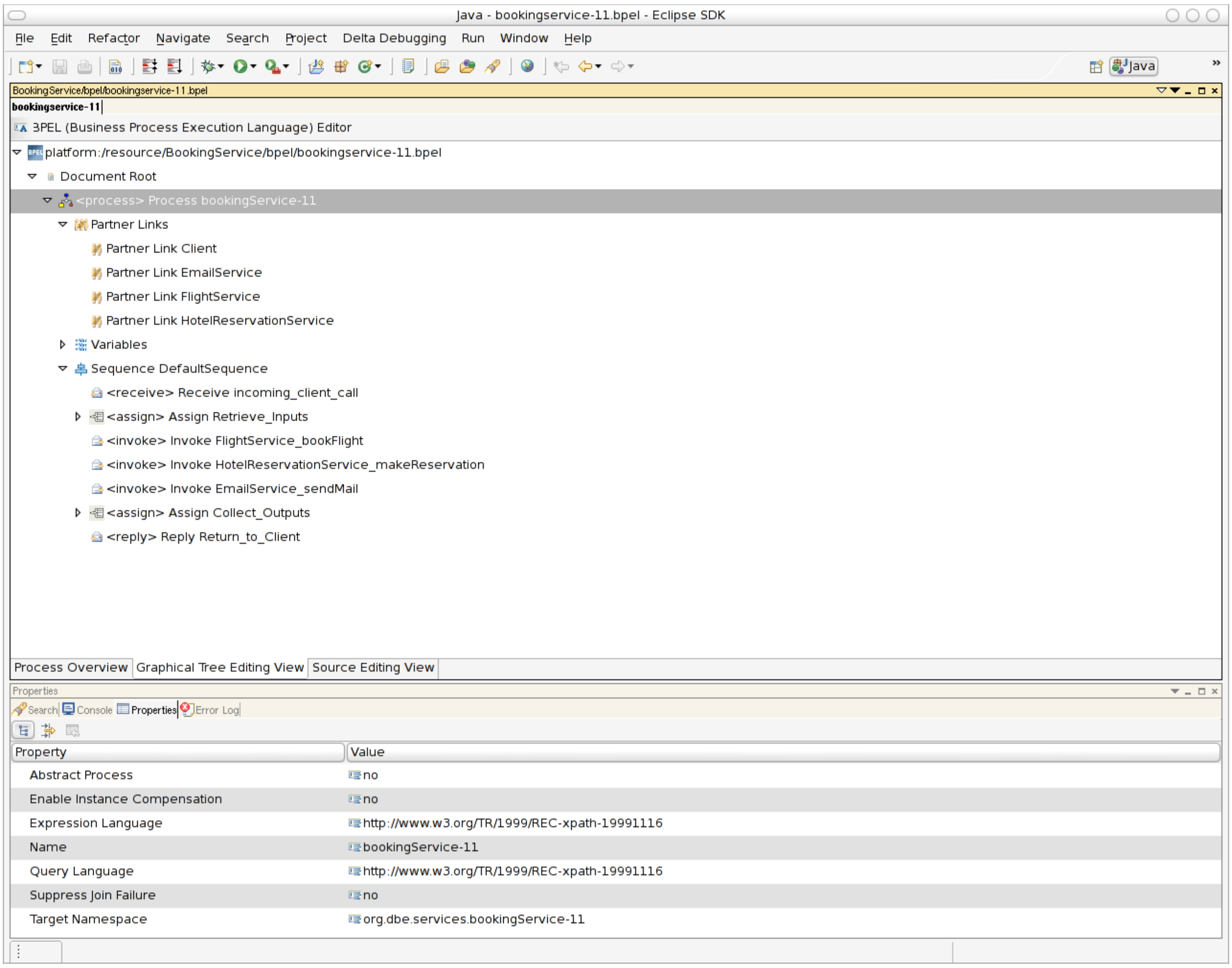


Illustration 24: BPEL Editor: Graphical Tree Editing View

Within graphical tree editing view, new BPEL activities can be added by right click on either the root process element or any other sub element. Select 'Add New Workflow Element', and then select on require BPEL element. Once the new element is added to the process tree, its properties can be added via the properties view while highlighting the particular element by clicking on it within the tree view.
In addition, to the graphical tree editing view, users can edit a BPEL file with the source editing view. This view has standard textual editing capabilities. Copy, cut, paste and save operations can be performed by right clicking on the editing view.

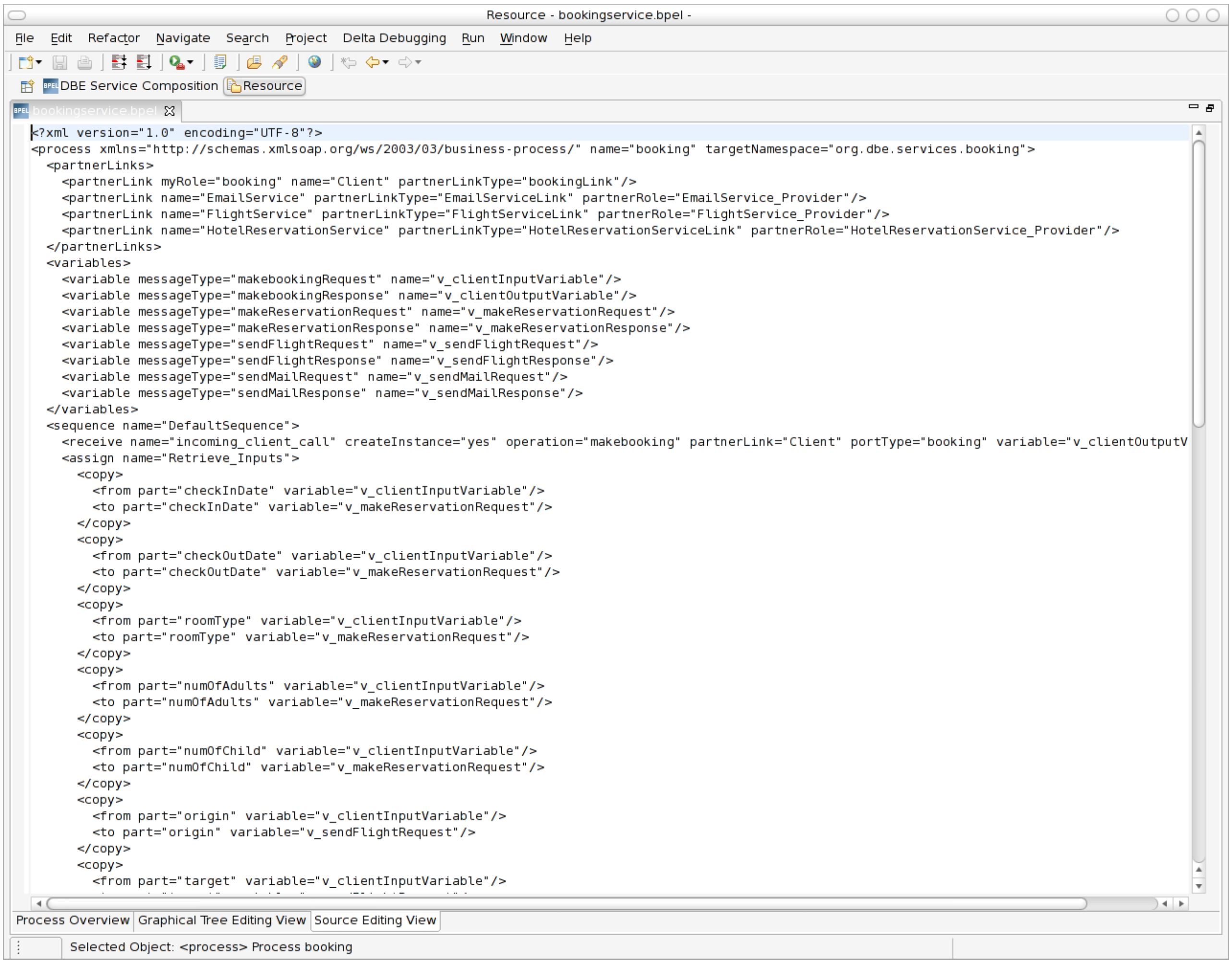


Illustration 25: BPEL Editor: Source Editing View