

**SIXTH FRAMEWORK PROGRAMME
INFORMATION SOCIETY TECHNOLOGIES**

IST-FP6-004559 SODIUM

“Service Oriented Development In a Unified fraMework”



D7: Specification of Visual Service Composition Language (VSCL)

Identifier

Date:	23/06/2005
Author(s):	Hjørdis Hoff, Roy Grønmo, David Skogan, Audun Strand
Distribution:	All partners
Workpackage:	WP2- Visual Service Composition
Version:	1.0
Status:	Final
Abstract:	This document presents the specification of the SODIUM Visual Service Composition Language



Revision History

Version Identifier	Revision Outline	Revision Author
0.5	First draft	Hjørdis Hoff & Roy Grønmo (SINTEF)
0.6	Textual corrections Corrected due to comments from other partners	Hjørdis Hoff & Roy Grønmo (SINTEF)
0.7	Model and text adjustments due to comments, considerations and new ideas.	Hjørdis Hoff (SINTEF)
0.9	Improvements of the document due to comments /input from other partners.	Hjørdis Hoff (SINTEF)
1.0	Final corrections and improvements have been made.	Hjørdis Hoff (SINTEF) and Roy Grønmo (SINTEF)



Executive Summary

This document contains the specification of the Visual Service Composition Language (VSCL) which is a graphical composition language. The intended use of the language is for defining service compositions containing heterogeneous (Web, Grip, P2P etc.) services. The language will be supported by the SODIUM Visual Editor.

The pilot requirement specifications (deliverable D5) and especially the requirement specification for the SODIUM platform (deliverable D3) have served as input to this specification. In addition, state-of-the-art analysis and literature search have been done for the three service domains (web, grid and P2P services) in order to determine the contents of VSCL.

The VSCL for a composite service is input to the SODIUM transformation engine. This engine translates VSCL to USQL for service discovery and querying for existing services in registries. The engine will also translate VSCL to USCL when existing services are identified. USCL is the lexical counterpart of VSCL which can be/is executed by the SODIUM execution engine.

Compositions are modelled as processes, whose structure defines the data and control flow dependencies between the service invocations. The language also includes a simple model of the interface of the component services, which – in case of Web services – can be automatically abstracted from a WSDL document.



Table of contents

1. INTRODUCTION	9
2. BACKGROUND.....	11
2.1 Model driven development	11
2.2 Service Oriented Architecture.....	11
2.3 Services	12
2.4 Service composition	12
3. VSCL REQUIREMENTS AND DESIGN CRITERIA	15
3.1 Requirements	15
3.2 Success criteria.....	16
4. SERVICE COMPOSITION SCENARIOS	17
5. THE VSCL BEHAVIOURAL MODELLING	19
5.1 Graph nodes	20
5.1.1 Tasks and concrete services.....	20
5.1.2 Composite tasks	24
5.1.3 Control nodes	24
5.1.4 Object nodes	25
5.1.5 Event nodes.....	26
5.1.6 Transformation node	26
5.2 Flow in the service composition.....	26
5.2.1 Object flow.....	27
5.2.2 Control flow.....	30
5.2.3 Event flow	32
5.3 Loop	32
5.3.1 Loop Node.....	32
5.3.2 For each	33
5.4 Discriminator	35
5.5 Conversations	36
5.5.1 Example.....	36
5.6 Quality of Service (QoS)	38
5.6.1 QoS Characteristics	38
5.6.2 QoS Requirements.....	39
5.6.3 QoS Offered	40



5.7 Transformation41
5.7.1 Discussion 41
5.7.2 Suggested approach and major challenges 45

5.8 Semantics46

6. STATIC STRUCTURE.....48

6.1 Web service interface for the composition48

6.2 Component diagram presenting for the interface relationships of the composition services49

7. CONCLUSIONS AND FURTHER WORK50

8. REFERENCES51

9. ANNEX A – REQUIREMENTS AND PATTERNS53



List of Figures

FIGURE 1 SERVICE ORIENTED ARCHITECTURE.....	12
FIGURE 2 COMPOSITION ABSTRACTION LEVELS	13
FIGURE 3 ABSTRACT VERSUS CONCRETE SERVICE COMPOSITION.....	14
FIGURE 4 PERSON ACTOR AND ROLES.....	17
FIGURE 5 COMPOSITION SCENARIO EXAMPLE.....	18
FIGURE 6 COMPOSITION (METAMODEL)	19
FIGURE 7 RELATIONSHIP BETWEEN TASKS AND SERVICES (METAMODEL)	20
FIGURE 8 FROM TASK MODEL TO CONCRETE SERVICE MODEL.....	21
FIGURE 9 TASK MODELLING	21
FIGURE 10 CONCRETE SERVICE WITH DATA TRANSFORMATIONS ON INPUT AND OUTPUT.....	22
FIGURE 11 SCALABILITY WITH A LARGE NUMBER OF CANDIDATE SERVICES.....	23
FIGURE 12 WEB SERVICE OPERATION.....	23
FIGURE 13 P2P SERVICE OPERATION	24
FIGURE 14 GRID SERVICE OPERATION	24
FIGURE 15 SUBTASKS (METAMODEL).....	24
FIGURE 16 CONTROL NODES (METAMODEL).....	25
FIGURE 17 OBJECTNODE (METAMODEL)	25
FIGURE 18 EVENTS (METAMODEL).....	26
FIGURE 19 TRANSFORMATION NODE (METAMODEL).....	26
FIGURE 20 FLOWS (METAMODEL).....	27
FIGURE 21 OBJECT FLOW CONSTRAINT (METAMODEL).....	27
FIGURE 22 OBJECT AND CONTROL FLOW IN UML 1.5.....	28
FIGURE 23 OBJECT FLOW WITH IMPLICIT CONTROL FLOW IN UML 2.0.....	28
FIGURE 24 PIN NOTATION FOR SIMPLIFIED OBJECT FLOW	28
FIGURE 25 UPDATES OF REGULAR DATA OBJECTS ARE ILLEGAL.....	28
FIGURE 26 LIST OBJECTS: THE READING PROBLEM.....	29
FIGURE 27 LIST OBJECTS: UPDATES INSIDE SUB COMPOSITION, READING OUTSIDE SUB COMPOSITION	29
FIGURE 28 CONTROLFLOW CONSTRAINT (METAMODEL).....	30
FIGURE 29 SEQUENCE (METAMODEL).....	30
FIGURE 30 EXCLUSIVE OR (CHOICE, XOR-SPLIT)	30
FIGURE 31 MERGE (OR-JOIN).....	31
FIGURE 32 FORK (AND SPLIT).....	31
FIGURE 33 MERGE.....	31
FIGURE 34 EVENTFLOW CONSTRAINT (METAMODEL)	32
FIGURE 35 TIME EVENT AND INTERRUPT (EXAMPLE).....	32
FIGURE 36 LOOP EXAMPLE: LOOP NODE (STEREOTYPED UML SUBACTIVITY)	33
FIGURE 37 LOOP EXAMPLE: <i>FOR EACH</i> (UML EXPANSION REGION)	34



FIGURE 38 LOOP- FOR EACH EXAMPLE (DATA MODEL).....	35
FIGURE 39 ALTERNATIVES AS SUB-ACTIONS.....	35
FIGURE 40 DISCRIMINATOR (METAMODEL)	35
FIGURE 41 SERVICE COMPOSITION AND CONVERSATION (TAKEN FROM [28])	36
FIGURE 42 CONVERSATION EXAMPLE (BUSINESS PROCESS MODEL)	37
FIGURE 43 CONVERSATION EXAMPLE (COMPONENT DIAGRAM).....	38
FIGURE 44 QUALITY OF SERVICE	38
FIGURE 45 MODELLING QOS CHARACTERISTICS	39
FIGURE 46 MODELLING QOS REQUIREMENTS	40
FIGURE 47 COMPOSITION MODEL WITH ACTUAL QOS VALUES	41
FIGURE 48 DATA TRANSFORMATION (METAMODEL).....	41
FIGURE 49 DATA TRANSFORMATION NEEDED	43
FIGURE 50 ALTERNATIVES FOR MODELLING DATA TRANSFORMATIONS.....	44
FIGURE 51 XSLT VS. QVT	45
FIGURE 52 SEMANTIC INFORMATION (METAMODEL)	46
FIGURE 53 IDENTIFYING ONTOLOGY CONCEPTS FOR THE INPUTS AND OUTPUTS OF A SERVICE	47
FIGURE 54 THE OWL-S CONGOEXPRESS EXAMPLE MODELLED IN UML.....	48
FIGURE 55 SERVICE INTERFACE EXAMPLE	48
FIGURE 56 INTERFACE RELATIONSHIP EXAMPLE	49



List of Abbreviations

BPDM	Business Process Definition Metamodel
BPMN	Business Process Modeling Notation
BPML	Business Process Modeling Language
EDOC	Enterprise Distributed Object Computing
MDA	Model Driven Architecture
OMG	Object Management Group
OWL	Ontology Web Language
P2P	Peer to Peer
PIM	Platform Independent Model
PSM	Platform Specific Model
QoS	Quality of Service
QVT	Query View Transformation
SOA	Service Oriented Architecture
SQL	Structured Query Language
UML	Unified Modeling Language
USCL	Unified Service Composition Language
USQL	Unified Service Query Language
VSCL	Visual Service Composition Language
WSDL	Web Service Description Language
WSFL	Web Service Flow Language
WSMO	Web Service Modeling Ontology
W3C	Wide World Web Consortium
XML	Extensible Markup Language

1. Introduction

As the number of available web services is steadily increasing, there is a growing interest for reusing basic web services in new, composite web services. There is a need for methods and techniques for federating basic web services into composite web services in a more adaptable manner than traditional programming. A service composition consists of a set of services and their descriptions and rules and requirements for their interaction with respect to data and control flow, quality of service, in/out parameters correspondence and mapping. Service Compositions may be defined in a lexical or graphical notation. Graphical is often easier for humans, while machines prefer textual. The textual descriptions may be used as scripts for process execution engines.

There are several proposed languages that can be used to model or specify different aspects of web service compositions. These are some of the most promising approaches:

- *UML 2.0 activity diagrams*. A graphical language which can be used to model control flow and data flow. No specialized support for service composition. Can be used quite freely and models may in general be produced with unclear semantics and the user may define models which are imprecise, such as with missing types for a data object. It has been shown that UML activity diagrams can be used to model web service compositions [1, 2], but it requires some guidelines and UML extensions to be used appropriately.
- *UML 2.0 sequence diagrams* [1, 2]. A graphical language which can be used to model control flow and data flow. It is unlikely that these alone can be used to fully model the aspects needed in a service composition. This is mainly due to the limited possibilities of parallel control flow and the focus of message view, where we in our service composition will focus more on a service view.
- *UML 2.0 communication diagrams* [1, 2]. A graphical language that focus on roles and how they interact. A link is established between the interacting roles, and all the messages they exchange are attached to that link. The messages are numbered in order to see the sequence of interaction which also gives an implicit time sequence. We claim that this diagram is not suitable for defining complex compositions since only sequential message flow is possible to describe (e.g. no parallel split or choice, no loops). The focus of this diagram is different from our service composition focus.
- *WS-BPEL / BPEL4WS* [3]. An XML notation that is the leading specification for executable web service compositions. It has been criticized for the lack of semantics [4] associated with the constructions, that there are unnecessary many syntactic ways to achieve the same semantic constructions (REF: SODIUM D6-USCL).
- *BPMN* [5]. A graphical notation that has been shown to be at least as good as UML 2.0 activity diagrams for modeling control flow patterns [6]. It has some appealing graphical notations that make it intuitive to understand. It is unsuited to be used for defining service composition in the current version due to lack of precise data object and data flow modeling.
- *BPDM* [7]. A graphical notation which is a profile of UML 2.0 component and activity diagrams. It is an interesting candidate which it remains to investigate fully. Although BPDM has been defined as a UML 2.0 profile, they also promote a specialized notation that goes beyond UML. It remains to investigate if there are existing tools for the specialized notation.
- *UML for EDOC* [8]. A graphical UML 1.5 notation. It is a profile that targets service-oriented modeling. It needs to be upgraded to UML 2.0 level.
- *UML for EAI* [9]. To be investigated.
- *JOpera Visual Composition Language JVCL* [10]. A graphical notation delivered by the SODIUM partner, ETH. It is suitable for doing service composition modeling. A limitation is that it is tied to one execution platform.



- *OWL-S [11] and WSMO [12]*. These are the two leading semantic web languages. Both are textual, OWL-S uses UML, and WSMO using a logical language. They are introduced to cover the description of semantics of Web services. In addition they also describe (at least OWL-S) the composition itself and thus has a great overlap with BPEL4WS.
- *GSFL [13]*. An XML notation for defining Grid service compositions. To be investigated.

There are some limitations with the existing proposals that we want to address with the introduction of yet another composition language – The Visual Service Composition Language (VSCL). First of all, many of the proposals use low-level textual notations in XML to represent the composition. A higher level graphical language has the promise to improve the development and maintenance of service compositions. There is not yet any de-facto standard, although BPEL4WS is close to becoming one. BPEL4WS is also likely to be modified and updated radically in the near future. Thus we need a composition language that is independent of a chosen execution language and that can be transformed into several languages depending on the user preference. This will make the language more stable for the users, and still adaptable to the ever-changing execution environments. Another limitation of the existing proposals is that they focus on a single service type, typically Web, Grid or P2P services. There is a need for a language which can be applied to any service type as well as combinations such as a composition combining several service types. The graphical languages mentioned above are more general purpose and they lack a specialized apparatus suitable for modeling executable service compositions. The goal is that VSCL can become a leading graphical service composition language with defined transformations from and to the most promising textual notations for service-oriented architectures such as BPEL4WS, WSCL, OWL-S, WSMO, SODIUM USQL and SODIUM USCL.

The goal is to implement the VSCL language in a new tool, but it is uncertain how far we get in the SODIUM project. So until we have such a solution in place, we have decided to use UML 2.0 as a tool where we can model service compositions with similar semantics as those described by VSCL. This means that we will define how to use UML for this purpose and which new UML extensions we propose to use for missing VSCL support constructions. While this pragmatic strategy means that the work on defining the SODIUM pilot specifications can be started immediately, it also means that the proposed optimal solutions of VSCL requires the development of a new tool. All the major limitations of UML will be noted down and possibly be reported back to the Object Management Group as suggestions for UML profiles and/or future improved UML versions.

This report is structured as follows; Section 2 presents some background information on model-driven development, service-oriented architecture what we mean by service composition modeling; Section 3 presents the requirements to VSCL; Section 4 presents typical use case scenarios of the VSCL language; Section 5 presents the key elements of VSCL and its draft technical solutions; Section 6 summarizes the VSCL language with the overall conceptual metamodel; and Section 7 covers the conclusions and future work

2. Background

2.1 Model driven development

Model-Driven Development (MDA) is defined by OMG. [14] gives the following overview of MDA:

The Model-Driven Architecture starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform.

MDA provides an approach for, and enables tools to be provided for:

- *specifying a system independently of the platform that supports it,*
- *specifying platforms,*
- *choosing a particular platform for the system, and*
- *transforming the system specification into one for a particular platform.*

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns.

The different kinds of models are the computation independent model (CIM), the Platform Independent Model (PIM), the Platform Specific Model (PSM), and the Platform Model. Model transformation is the process of converting one model to another model of the same system.

2.2 Service Oriented Architecture

According to W3C [15], a Service Oriented Architecture (SOA) specifies a set of components whose interfaces can be described, published, discovered and invoked over a network. SOA vies to promote software development in a way that leverages the construction of dynamic systems which can easily adapt to volatile environments and be maintained. The decoupling of system constituent parts enables the re-configuration of system components according to the end-user's needs and the system's environment. Furthermore, the use of widely accepted standards and protocols that are based on XML and operate above internet standards (HTTP, SMTP, etc) enhances interoperability [16].

Any service-oriented environment is expected to support several basic activities [TP 2002]:

- Service creation
- Service description
- Service publishing to Intranet or Internet repositories for potential users to locate
- Service discovery by potential users
- Service invocation, binding

A service is unpublished when it is no longer available or needed, or in case it has to be updated to satisfy new requirements.

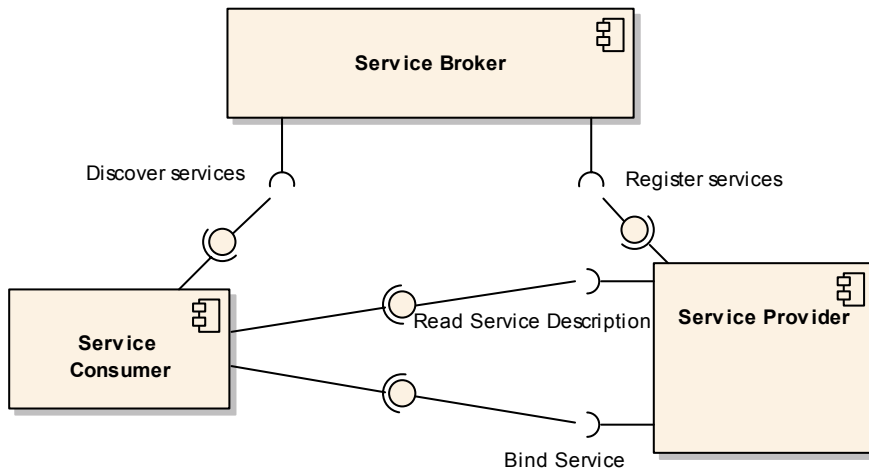


Figure 1 Service oriented architecture

In a Service Oriented Architecture (SOA), there are three main groups of participants/roles which are depicted in Figure 1:

- the Service Providers, a service owner that offers services to Service Consumers by publishing them with a Service Broker. The Service Provider offers two interfaces: one for reading service descriptions and one for service binding;
- the Service Consumers, a service client that needs to discover which available services may perform a certain task, select one or more services, connect to the Service Provider and execute the service(s);
- the Service Brokers, is mediating between Service Providers and Service Consumers, and offers the possibility for Service Providers to publicize their available services, and offer the possibility for Service Consumers to discover available services and gives them binding information. The Service Broker offers two interfaces: one for service discovery and one for service registration

Which interfaces are used by which components are depicted by the interface – socket relationship.

For Web Services, the instantiated SOA called Web Service Architecture will consist of:

- UDDI as service broker offering to publish and discovery possibilities.
- WSDL for service descriptions
- XML for messaging between the client and the service.

2.3 Services

The three different types of services addressed in SODIUM are: Web Services, Grid Services and Peer-to-Peer Services. These are presented in more detail in deliverable D4: “Generic Service Model Specification”.

2.4 Service composition

This section describes different composition abstraction levels and defines service composition modelling within this context. Figure 2 shows different abstract levels for modelling, describing and executing compositions. The most abstract level is the business process modelling where the high level processes involved are modelled without details about concrete services or data flow. The business process model contains processes that can be decomposed as compositions themselves and where some processes may be manual and others automatic. *The service composition is the detailing of processes that can be automated and where the service tasks are detailed with respect to data flow and where each individual task can be performed by a single service.* The process

execution is handled by an execution engine that takes care of control flow, data flow and data transformations. Service description describes the service composition as a whole. The service implementation is the realisation of the service composition behaviour on a concrete platform with one or more concrete protocol bindings.

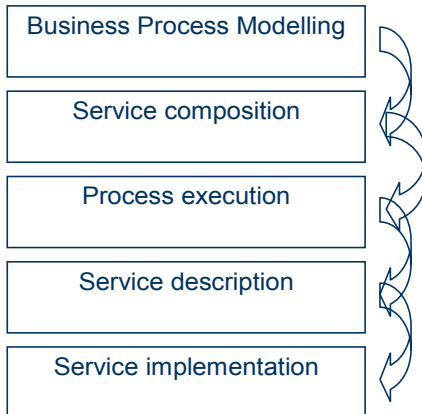


Figure 2 Composition abstraction levels

Figure 3 shows more details about the service composition layer which is the focus of the VSCL specification. We distinguish between the *abstract composition* and the *concrete composition*. In the abstract composition no choices about concrete services are taken. But the tasks are precisely defined with the requested inputs and outputs, with a possible future extension to VSCL being the possibly to specify *effects* of each desired task. The inputs and outputs may use semantic descriptors such as references to well-known ontologies. Furthermore there may be QoS requirements defined. It must be possible to automatically generate a USQL document for querying existing service registries about concrete candidate services. These candidate services can then be used to define a concrete composition with one chosen or a set of alternative services for each task in the abstract composition. The concrete composition must be expressive enough so that complete USCL documents can be generated. The USCL document can then be sent to an execution engine such as SODIUM Execution Engine for process execution. An extended BPEL document [3] could also be generated as an alternative to USCL. It is expected that extensions of BPEL are needed in order to meet the requirements of SODIUM such as invocation of other services than web services (P2P, Grid,...). It is outside the scope of SODIUM to implement a BPEL execution engine that handles the SODIUM-specific extensions.

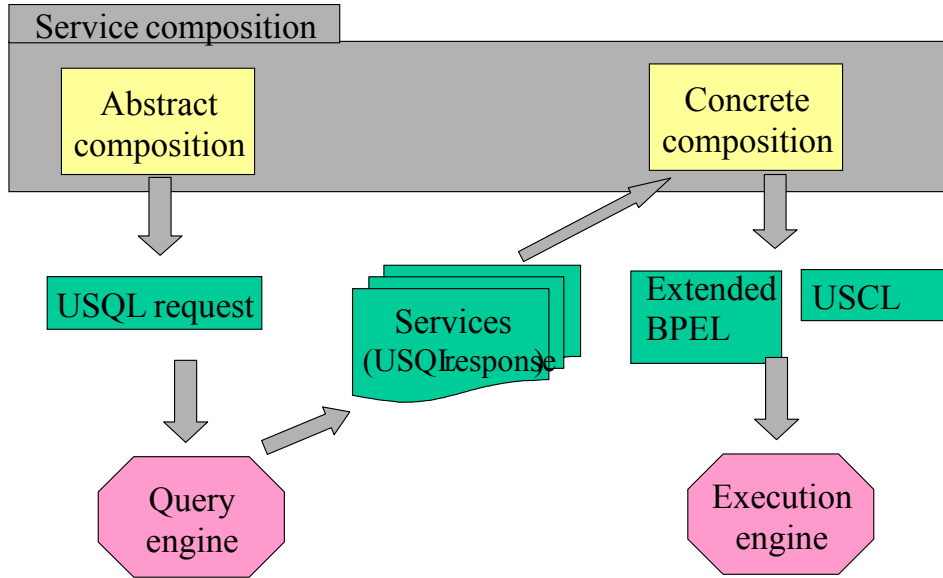


Figure 3 Abstract versus concrete service composition

3. VSCL requirements and design criteria

This first section of this chapter gives an overview of requirements for the Visual Composition Language for heterogeneous service composition. The requirements are divided in two groups: the first group consisting of the requirements that shall be supported by the language, and the other group consisting of requirements that may be supported by the language.

The second of this chapter gives an overview of success criteria for the Visual Service Composition Language.

3.1 Requirements

This chapter gives an overview of the identified requirements and input from related work. Each of the requirements that will be included in/supported by VSCL is marked. In addition there is a comment which states why/why not the requirements will be supported.

The SODIUM ambition is to define a visual composition language that shall *at least* support:

- composition of heterogeneous services (web, grid and P2P services) and hide platform distinctive characteristics;
- modelling of abstract compositions where the actual services are not yet chosen;
- modelling of concrete compositions where the actual services are chosen;
- the needs of the pilot applications (SODIUM deliverables D3 and D5);
- basic control flow (Sequence, Parallel split/and-split, Synchronization/and-join, Exclusive choice/or-split, Simple merge/or-join);
- data flow and definition of data transformations;
- be expressive enough to be transformed into executable specifications in USCL;
- definition of the information needed to generate queries according to the USQL language
- definition of composition interface (service operation) and generation of web service description for it;
- definition of Quality of Service (QoS) requirements (a selected set of QoS attributes) for the individual services;
- semantic annotation of service operation, input/output parameters for the individual abstract tasks;
- discriminator pattern for the selected service to perform a specific task;
- generation of service queries¹ that are used for dynamic service discovery at execution time.

The visual composition language *may also* support:

- the possibility of defining Quality of Service (QoS) requirements for the composition as a whole;
- the possibility of adding Quality of Service (QoS) offered values to the selected services in a concrete service description;
- semantic definitions integrated in the composition such that inputs and outputs can be precisely defined, and that the USQL descriptions can be fully generated, and that information equivalent to OWL-S can be captured.
- control flow loops and for each;

¹ USQL statement included in the generated USCL document.



- conversation pattern[17]

The visual composition language *will not cover*:

- service interface modeling and service dependency modeling
- sequencing of operations in the service interface model (e.g. Login() before Order());
- modeling of fault and compensation-handling;
- modeling security requirements for given services, like e.g. message encryption strategy and permissions;
- various kinds of patterns:
 - Advanced branching and synchronization patterns (sources [18-20])
 - Structural patterns (sources [18-20] ,([21]))
 - Patterns involving Multiple Instances (sources [18-20])
 - Cancellation patterns (sources [18-20])
 - Inter-workflow synchronisation (source [18])
 - State-based patterns (sources [18-20] ,([21]))
 - Temporal relations (sources [18]).

At the moment, the ambition level might seem high. If or when the “may also support” requirements will be considered frequently during the rest of the project.

3.2 Success criteria

We have made some choices for the VSCL language which is based upon the goal to satisfy a number of success criteria listed here:

- **Expressiveness.** The language shall be expressive enough to meet the requirements listed in the previous section.
- **Ease-of-use.** VSCL models shall be easy to understand for experienced modelers. This means that it shall be easy to define the service compositions, easy to understand the compositions, it should be intuitive to interpret the semantics of the notations.
- **Executable.** It must be possible to define a model with enough details so that a complete executable document (USCL, BPEL++) may be generated from it.
- **Independence of executable language.** The VSCL language shall be independent of a particular execution language. The motivation is that we do not want to be tied to one language, especially when there are many competing execution language proposals.
- **Task modeling support.** It must be possible to specify tasks for which there are not yet defined or discovered concrete services that can solve the task. The task description capabilities must be precise enough to capture semantics and QoS descriptions covered by the USQL language and optionally those covered by OWL-S and WSMO. It must also be able to connect tasks with concrete services such that one can regularly search for competing services solving the same task.
- **Information hiding and views.** In order to not overload the user with information it must be possible to present the VSCL models with different views depending on the working mode and current concern area.

4. Service composition scenarios

This chapter presents some user scenarios that exemplify how a Service composer (hereafter called *user*) goes about creating composed services. The Visual Editor is used for service composition. The user also gets access to the USQL Engine front-end for service discovery. The ServiceValidator is responsible for inspecting and quality assurance of the created compositions.

The person actor and roles are presented in Figure 4.

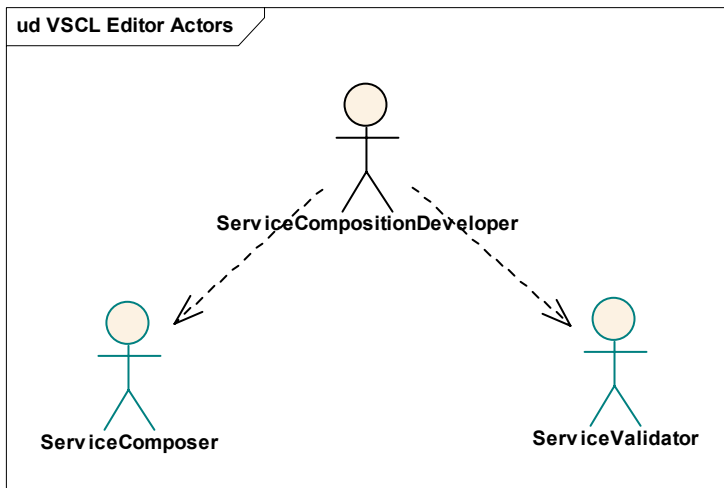


Figure 4 Person actor and roles

A number of aspects are covered by a service composition language. The service composition is a specification that consists of one or more services with a defined relationship. The relationship states which services communicate, how they communicates (order/flow of control – a number of possibilities) and what they communicate (flow of data/messages). This is defined as a graph with nodes and edges. In addition to the who, how and what, the composition covers the transformation of service output to service input, requirements on the communication (QoS and security) and what happens in case of failure. To further complicate it all, not all services in the composition are decided at specification time, but some might be selected at execution time!

A number of services may also be services that are not published in a global service registry like e.g. a UDDI-registry. These services may also be included in a composition.

The new service composition may be stored in a Repository (like the SODIUM Repository) and may be reused in other compositions.

Example: Creating a Service Composition

Service composition of pre-known services (see Figure 5, activity 2.1.2.2)

In the example scenario, the user has knowledge about which services he/she wants to include in a specific service composition, e.g. previous used services or services regulated by an already established agreement with a service provider. All the services in the composition are selected at specification time.

Service composition of not pre-known services (Figure 5, activity 2.1.2.1)

In the example scenario, the user has little or no knowledge about which services he/she wants to include in a specific service composition. He/she has little knowledge about available services, but has good knowledge of which tasks that might be performed by a service, e.g. previous used services or services regulated by an already established agreement with a service provider. The services in the composition are selected at specification time after service discovery by use of the SODIUM USQL Engine.

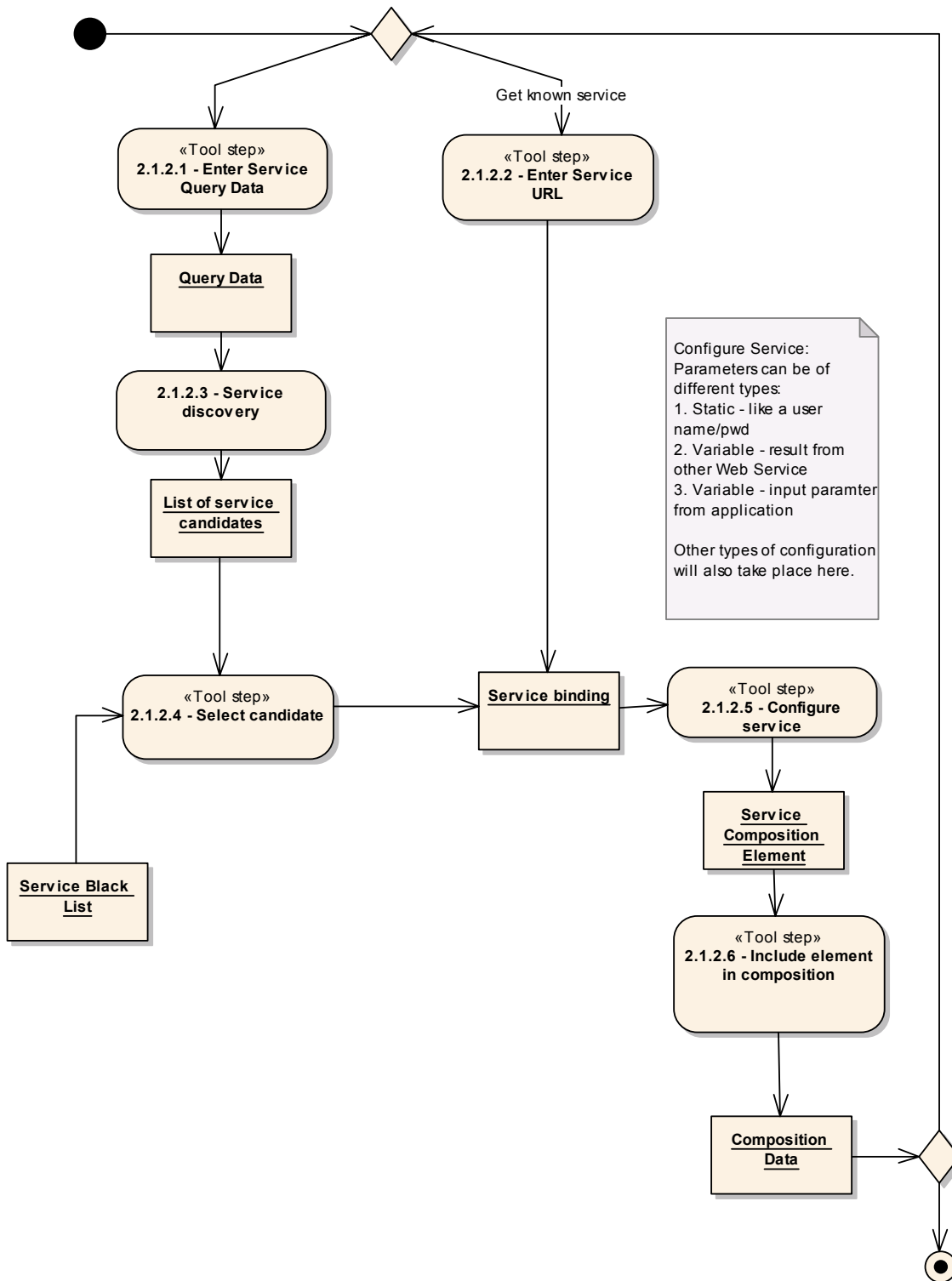


Figure 5 Composition scenario example

A third possibility (not included in *Figure 5*), is that the user does not select all the services at specification time, but that only an abstract specification of the service is defined. In this case, some of the services in the composition are selected at specification time after service discovery by use of the SODIUM USQL Engine, the rest is selected at execution time based on the abstract service specification.

5. The VSCL behavioural modelling

This chapter presents the concepts included in the SODIUM Visual Composition Language for behavioural modelling, and discussion and modelling examples of the different concepts in VSCL. Behavioural modelling concerns the modelling of what is to be done, by whom in which order. The purpose is to present what the VSCL contains in order to cover the requirements presented in chapter 3 to support a user creating heterogeneous service compositions. The models do not contain all details.

The concepts in the metamodel will be implemented and offered by the Visual Editor. How the metamodel will be implemented is presented in SODIUM project deliverable D9.

The VSCL metamodel is modelled in UML 2.0 [22-24]. The constraints are either described in natural language or using Object Constraint Language (OCL) [25, 26].

Please note that the model examples presented in this chapter serves as input to the Visual Editor specified in D9 and to be implemented in D12, which is one possible implementation of VSCL metamodel that will be based on UML2.

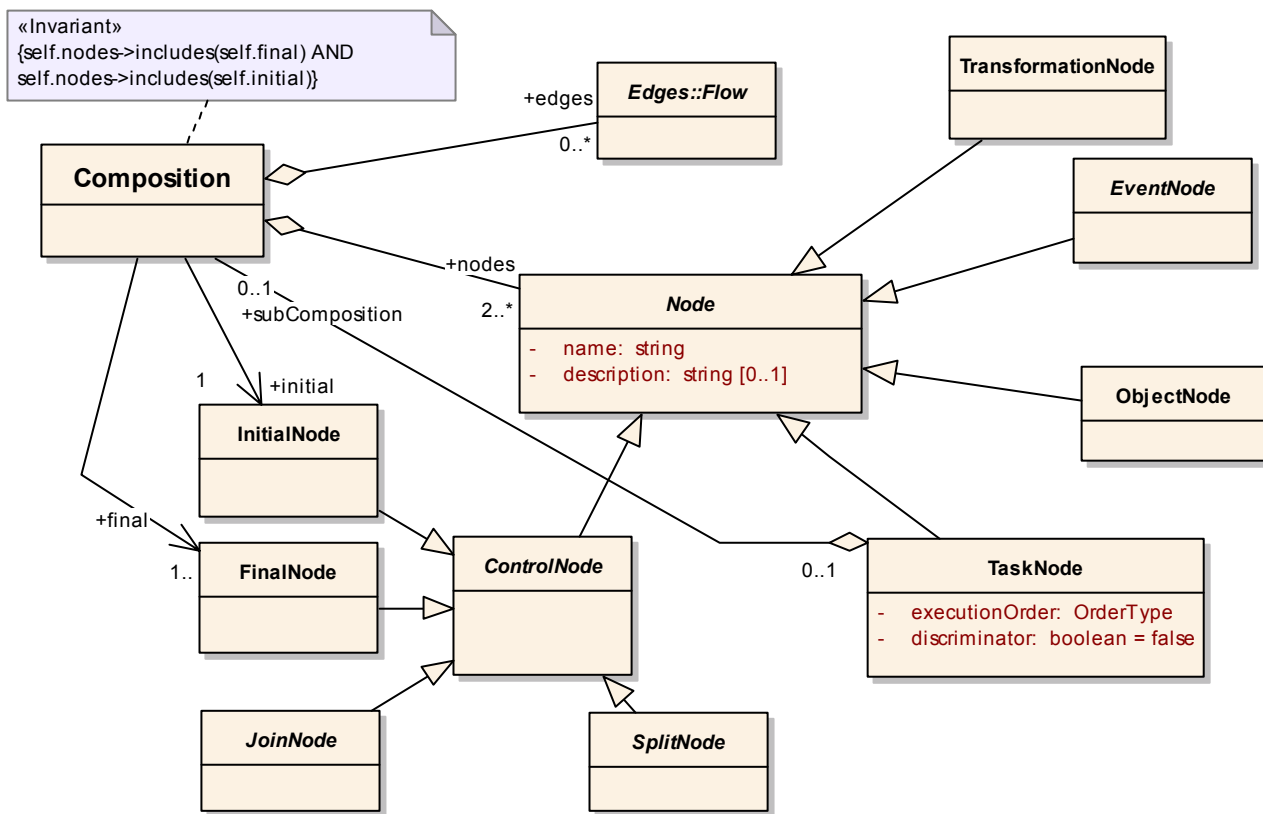


Figure 6 Composition (metamodel)

A service *Composition* consists of *Nodes* and *Flows/edges*. The *Nodes* are *TaskNodes*, *ControlNodes*, *ObjectNodes*, *EventNodes* or *TransformationNodes*, whereas the different kinds of *Flow*² are used to specify flow of control and data between nodes..

² Note that “Edges::” in the model refers to the *UML package* where *Flow* is defined.

5.1 Graph nodes

5.1.1 Tasks and concrete services

A composition consists of tasks that are to be performed in a specific order. The ordering is defined by the use of Flows and ControlNodes. A Task may be simple or composite. If composite, there exists a sub-composition for the Task. This gives a hierarchical composition similar to SubActivity in UML2.

Usually the composition developer model works in a top-down manner, meaning that the developer starts out by modelling a set of tasks (*TaskNode*) without deciding which service operations to actually execute them. The user then defines a task with a certain name, input and output parameters the ingoing and outgoing flows, and possibly also giving QoS requirements and associating various parts of the task definition to ontology concepts. This may be considered as being the abstract part of a Task. The user has of course the possibility of working “bottom up”. This is done when the developer has decided which service operation(s) to use in the composition. The composition will then still contain a Task with the selected service(s) that shall execute the Task.

So, if we go back to the top-down approach where the developer now has defined an abstract Task, the abstract task is used as input to a query engine used for discovery of suitable service candidates. When/if a service query is executed, the result populates the list of *candidateServiceOperations*.

A heterogeneous composition may consist of tasks executed by different kinds of services. The types defined here are: P2P, Web or Grid services. One or more services may be selected to realize/execute a specific task. When more than one service operation(s) are selected, the developer may state how the execution is to be done. There are three different possibilities. Either the services may be executed in parallel, sequential or in a random order, where the last two are (associated with a response time limit. The services in the *selectedServiceOperations* list are considered “equal” with respect to the service they provide, but other aspects like e.g. QoS characteristics may vary between them.

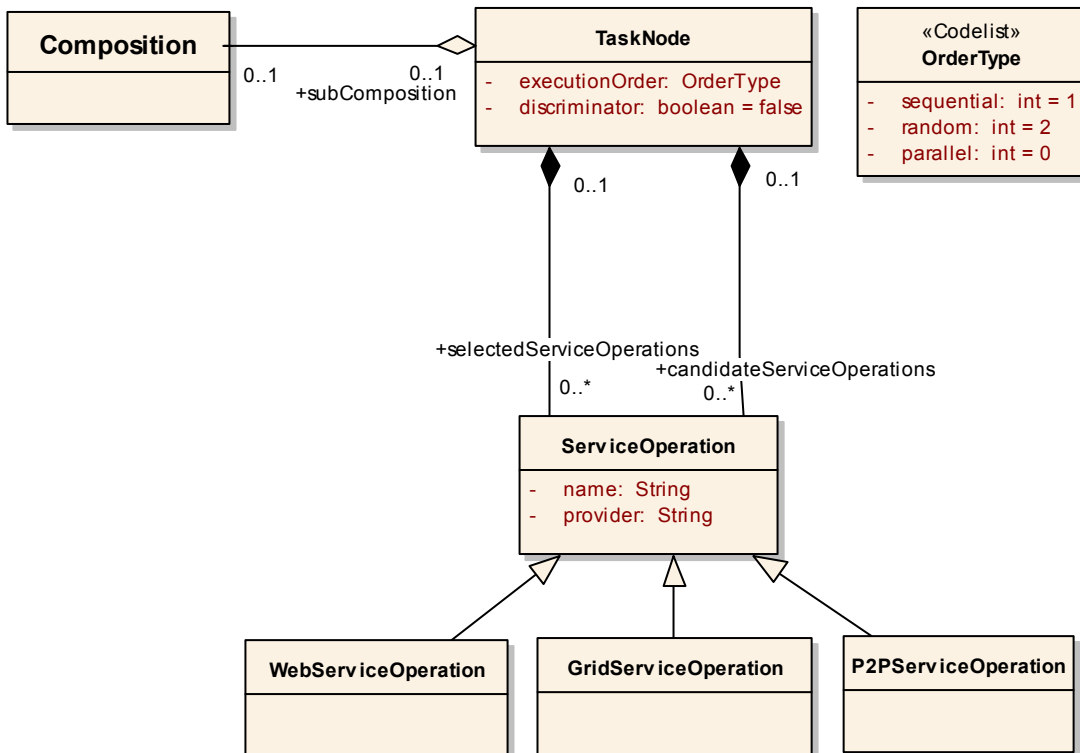


Figure 7 Relationship between tasks and services (metamodel)

For each task we precisely define its inputs and outputs by linking them to semantically defined ontology concepts and QoS requirements for each of the tasks. The information in the task model is

then used to search for existing services, see Figure 8. The discovered candidate services are then used to realize the different tasks of the task model and we ideally end up with a fully defined concrete service model with one (or possibly more) identified service for each task.

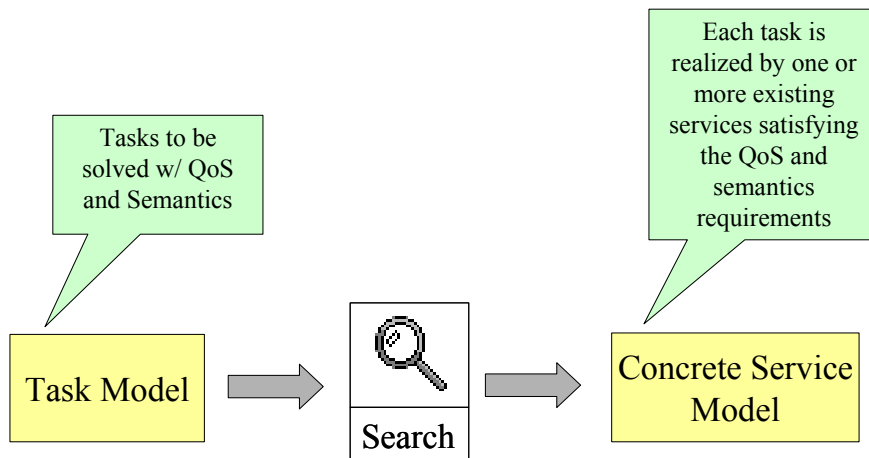


Figure 8 From task model to concrete service model

Figure 9 shows an extract of a task model. The input and output parameters of each task are defined and linked to an ontology concept. The tasks may additionally register preconditions and desired effects, but this possibility will not yet be included in VSCL. This is done to narrow the search to return only the most relevant services. There could also be registered QoS requirements that apply to the service composition as a whole, but this is outside the scope of SODIUM even though it is more likely that the user cares about the overall QoS than the individual QoS offerings for each task. It requires more computation by the search engine as it needs to consider how individual QoS offerings influence the composition as a whole.

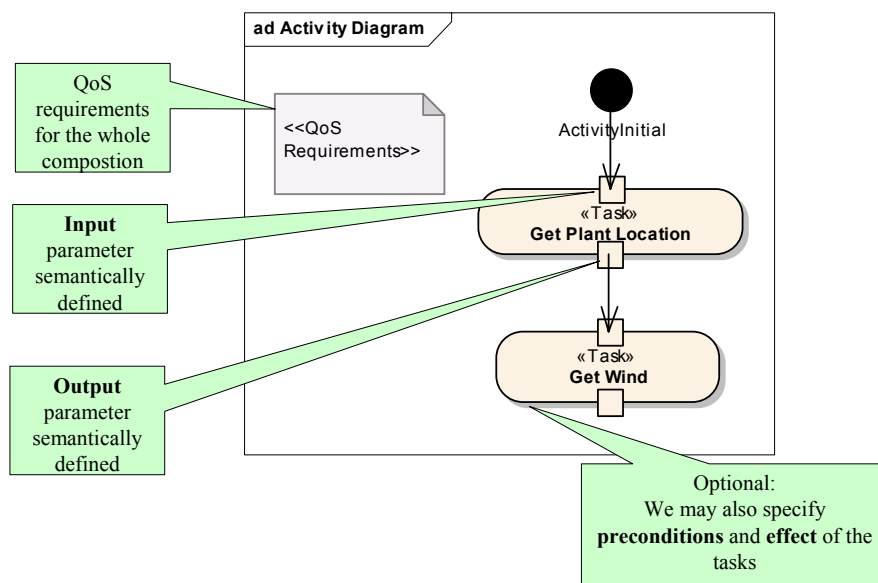


Figure 9 Task modelling

Each task can be realized by many existing services (Figure 10). If lucky, we find services that are perfect matches, meaning that the input and output parameters of the service are identical to the ones specified for the task. If they are not identical, transformations are defined between them. Note that when preconditions and effects are not included in the search, a manual inspection is needed to make sure that the service delivers what the service requester needs. The following example supporting this claim:

Let's say we have two services that both take a power plant name as input and deliver a coordinate location as output. The first service delivers the location of the plant itself, while the other service delivers the nearest airport location. Even though their defined input and output are syntactically identical, they do not deliver the same service.

In many cases we may find published services that are quite close to the requested ones. In such cases, simple data transformations may be enough to solve the difference; in other cases it may be wise to introduce new intermediate tasks. The metamodel supports treating services with only data transformations needed as candidates for realizing a task. An example is shown in Figure 10. Each concrete service shall have a descriptor of the service type (indicated by stereotype) and complete reference to an operation that can be fully executed. The service may optionally have semantic references for the data types used and QoS offered described depending on what kind of information we can retrieve for the service.

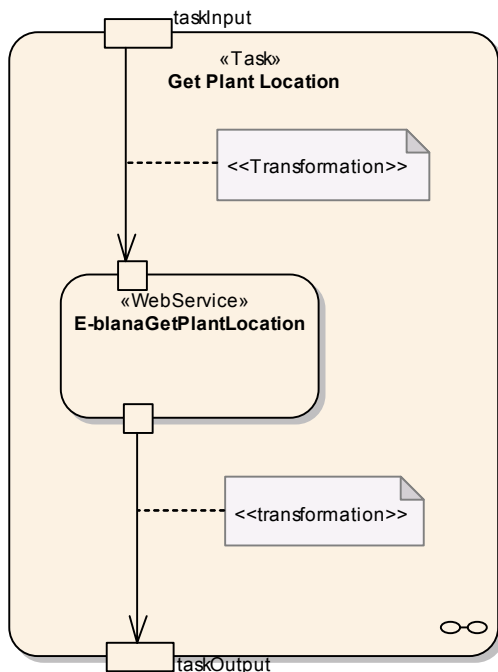


Figure 10 Concrete service with data transformations on input and output

The composition modelling and the searching for realizing services is an iterative process. This is further described in SODIUM Methodology, deliverable D19. The composition model needs to be adjusted according to the found services. There is also a need to repeat the searching for services regularly to ensure that the best services are found and used. Since the tasks are the basis for the searching, we need to keep the tasks in our model even if there are identified concrete services for the tasks. This will make the entire composition graph structure very complex which requires different views of the model. A *task view* is needed so that the composition modeller can focus on the tasks only. A *concrete service view* is needed to show the (possibly) executable composition with all the chosen services. If there are no found services for a task, then the task itself and an indication of missing services should be shown.

There may also be a large number of candidate services for each task which requires a scalable solution. Figure 11 shows a sketch of how this may be achieved. The modeller may click on a task in the task view requesting to see its corresponding candidate services. Each candidate service may be presented in a separate browser showing the status of each service. Those that have a status as *chosen* may also be given with more details as shown at the right part of the figure. This browsing functionality will require proper communication between the visual editor and the Unified Service Query Engine so that the list of candidate services is up-to-date.

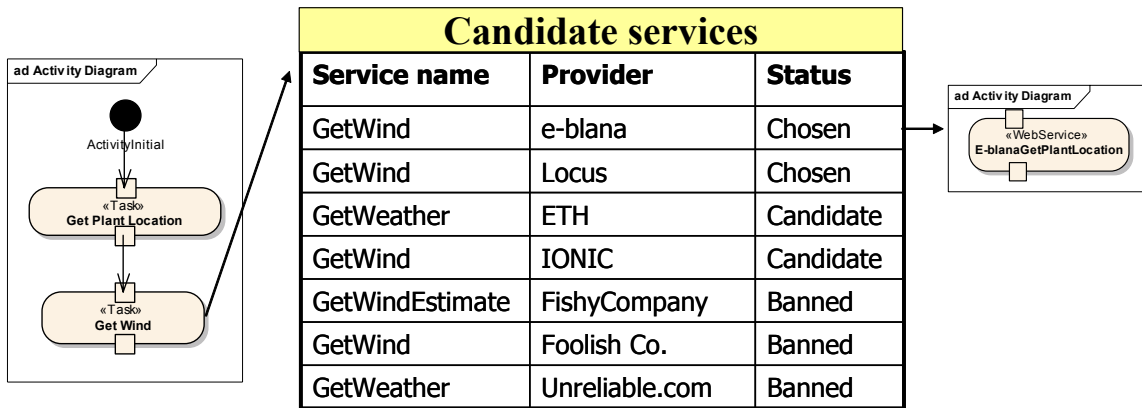


Figure 11 Scalability with a large number of candidate services

It is important that the modeller does not get mixed up with a task and the corresponding concrete services. In UML, e.g. stereotypes may be used to distinguish them (Task, WebService, P2PService and GridService). Another important requirement of the visual editor is flexibility. Therefore we must not be too restrictive for the modelling of tasks and concrete services. In some cases the modeller may know before searching one ore more of the services to use in the composition. In such cases it must be possible for the user to register the concrete service directly. The user may then choose to let the visual editor automatically define a corresponding task for the concrete service, or the user may choose to specify that this concrete service shall not be associated with a task. In the latter case no searches will be done for competing services, since only tasks are used as a basis to search for services. When the user defines a concrete service without a corresponding task, this should be explicitly registered and possibly viewed in the model by some kind of text or symbol showing that the service is fixed. Such fixed concrete services should probably appear in the task view model so that this model appears complete without missing “holes” for which it would be tempting to introduce new tasks.

5.1.1.1 Web Services

To represent a web service operation, activities are stereotyped *WebService*. A web service activity has a set of tagged values. The provider of the web service is defined by the tagged value *provider*. The URL to the WSDL file is registered in the tagged value *wSDL*. The exact service operation to invoke is given by the three tagged values *service*, *portType* and *operation*.

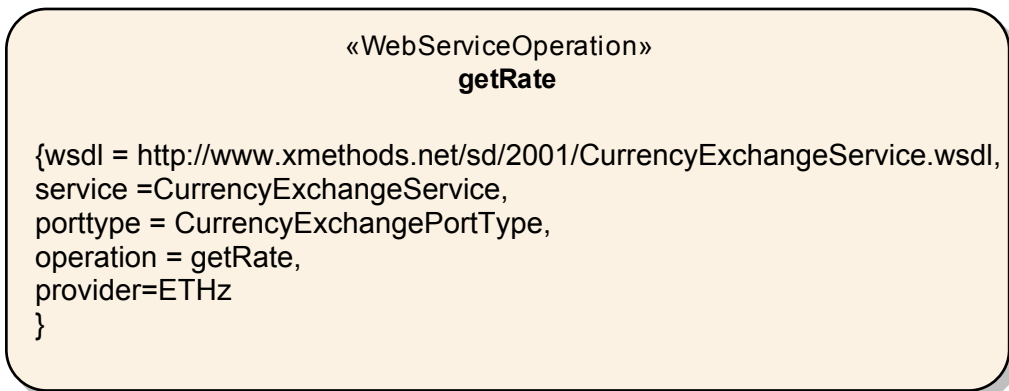


Figure 12 Web Service Operation

5.1.1.2 P2P

To represent a p2p service operation, activities are stereotyped *P2PServiceOperation*, Figure 13. The five tagged values *type*, *PSDL*, *Operation*, *service* and *Pipe* are sufficient to identify the p2p service operation to call.

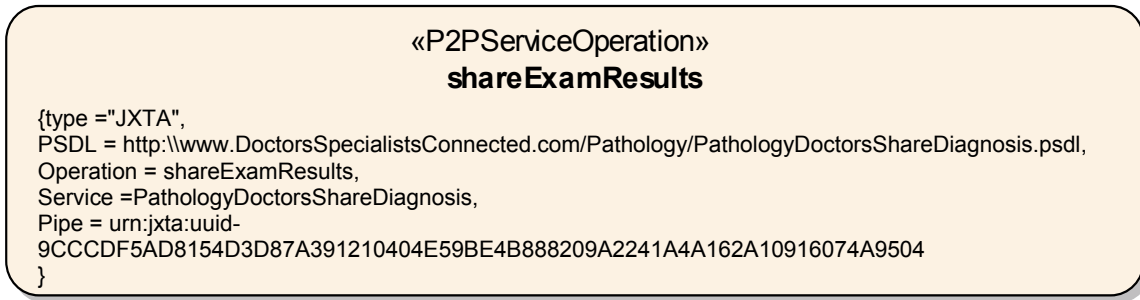


Figure 13 P2P Service Operation

5.1.1.3 Grid

To represent a grid service operation, activities are stereotyped `GridServiceOperation` Figure 14. The six tagged values `gwsdl`, `serviceLocation`, `service`, `operation`, `portType` and `resourceInstance` are sufficient to identify the grid service operation to call. GWSDL is an extension to WSDL defined as part of the Open Grid Services Infrastructure (OGSI). It is believed that GWSDL will be replaced by WSDL 2.0. GWSDL as well as OGSI are deprecated.

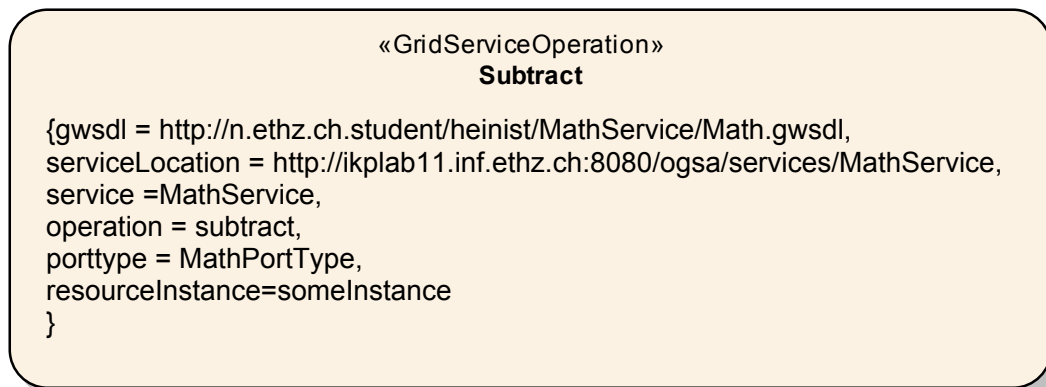


Figure 14 Grid Service Operation

5.1.2 Composite tasks

In some cases it is convenient to detail a specific task node into a sub composition (analogous to subactivity in UML 2.0).

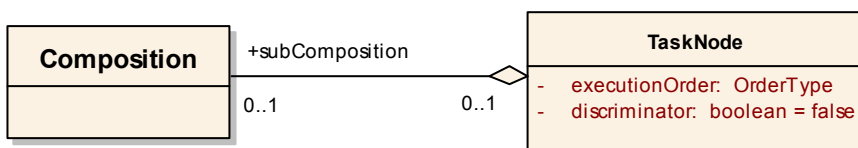


Figure 15 SubTasks (metamodel)

In this case the task will have a Composition associated with it through the subComposition aggregation. This aggregation relationship is shown in Figure 15.

5.1.3 Control nodes

Control nodes are composition nodes that represent specific crossings for control flow.

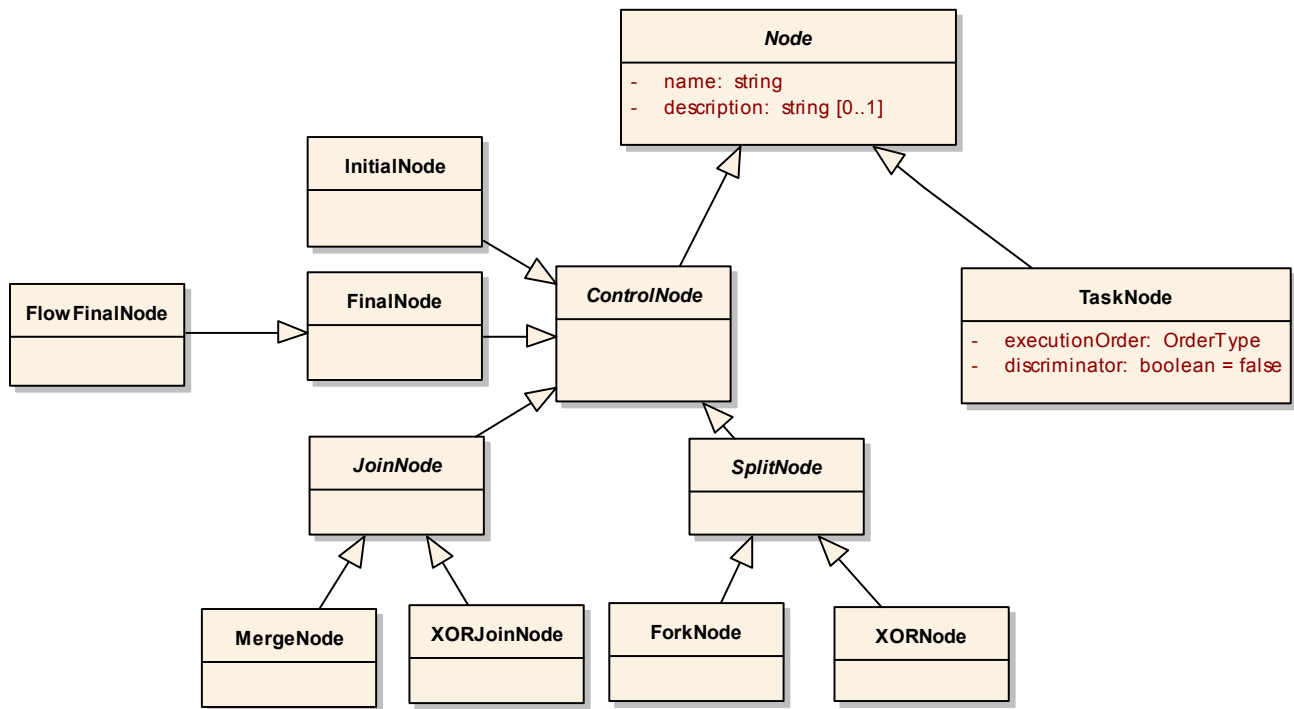


Figure 16 Control nodes (metamodel)

A number of control nodes are defined:

- *InitialNode* – which is the start node of a composition;
- *FinalNode* – a composition may have one or more end nodes; *FlowFinalNode* may be used for indicating the end of a specific flow (this will not be supported as due to lack of requirements)..
- *SplitNode* – which is either an exclusive OR (*XOR*) or a Fork node;
- *JoinNode*– which is either a *Merge* or a *Join* node.

Examples on use of the latter two are given in section 5.2.2.

5.1.4 Object nodes

An *ObjectNode* has one or more associated *DataObjects*, see Figure 17. The list of data objects is ordered. These data objects are used for data transfer between the tasks. A data object has a specific *ObjectType* (optional). Future extensions will consider different kinds of data objects such as transient, persistent within a composition, and persistent outside a composition.

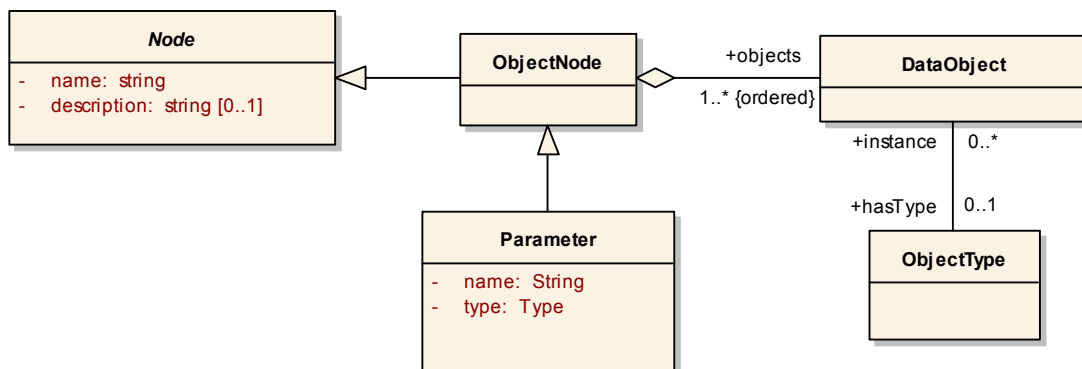


Figure 17 ObjectNode (metamodel)

A specialization of ObjectNode is Parameter which is presented in section 5.2.1.

5.1.5 Event nodes

EventNodes are nodes that take no input, but that overrides the current flow by giving control to a specific TaskNode based on specific criteria. For CancelEvents, this is e.g. the user giving a cancel execution signal to a specific task, whereas the TimeEvents occur at a specific point in time and may be used to start of a given task or composition at a given time, see Figure 18.

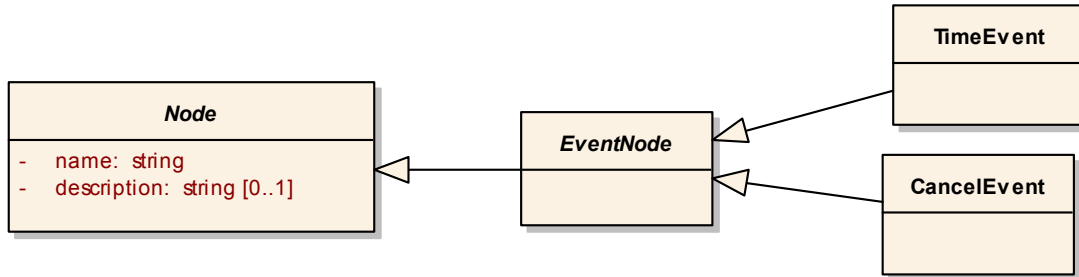


Figure 18 Events (metamodel)

See section also 5.2.3.

Events are on the “may be supported” list so whether or not it will actually be implemented will be decided in the autumn of 2005 and will in that case be further explored in the next version of D9.

5.1.6 Transformation node

Transformation nodes are used for defining data transformations as expressions in QVT [27]. The transformation node is used for one-to-many, many-to-one and many-to-many data transformations.

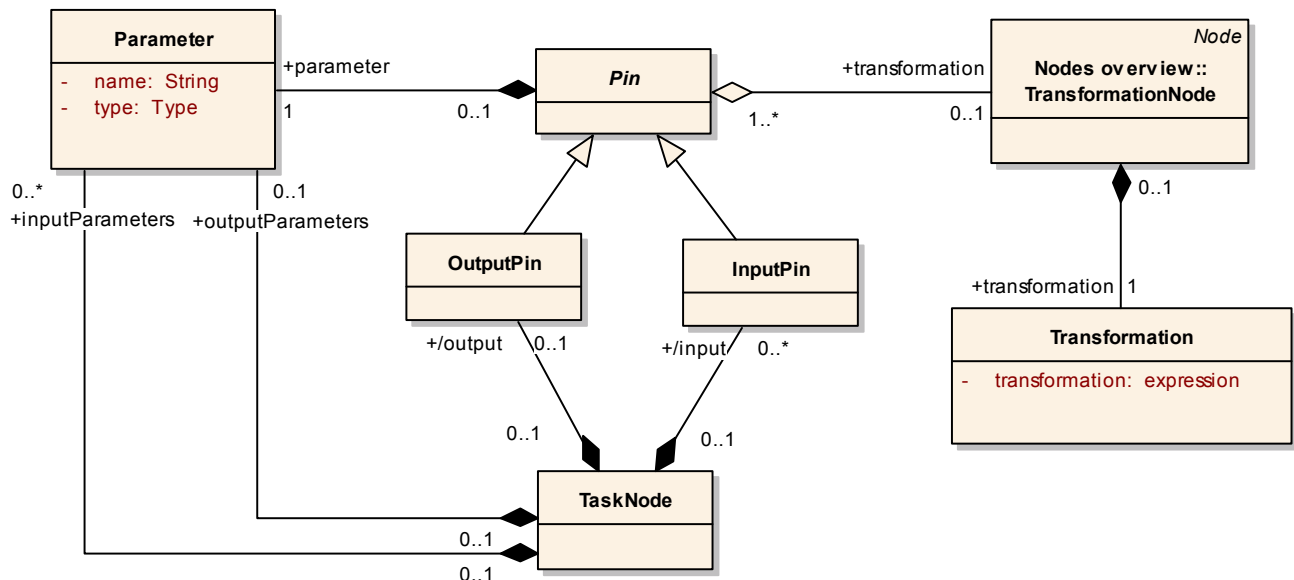


Figure 19 Transformation node (metamodel)

For one-to-one transformations another strategy is chosen. See section 5.7.

5.2 Flow in the service composition

A Flow indicates a directed flow of either flow of control (ControlFlow or EventFlow) or flow of data objects (ObjectFlow). A Flow has one source node and one target node. Further details may be found in the subsequent sections.

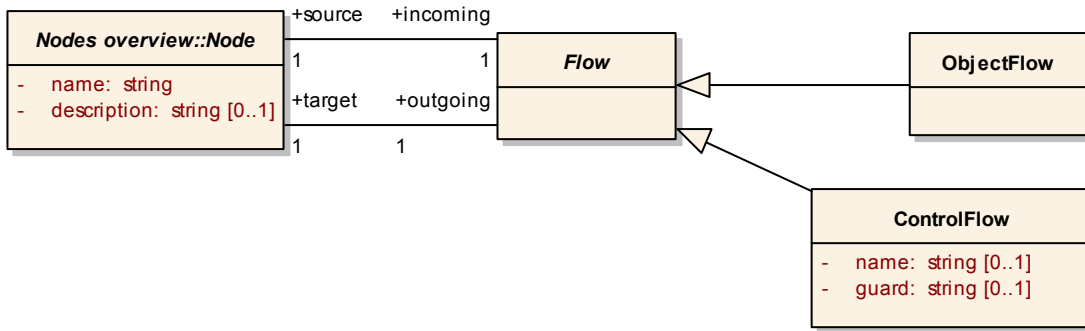


Figure 20 Flows (metamodel)

5.2.1 Object flow

As previously mentioned, the *ObjectFlow* represents the flow of data objects between a source node and a target node, these nodes being either two *ObjectNodes*, or an *ObjectNode* and a *TransformationNode*. The invariant is presented in Figure 21.

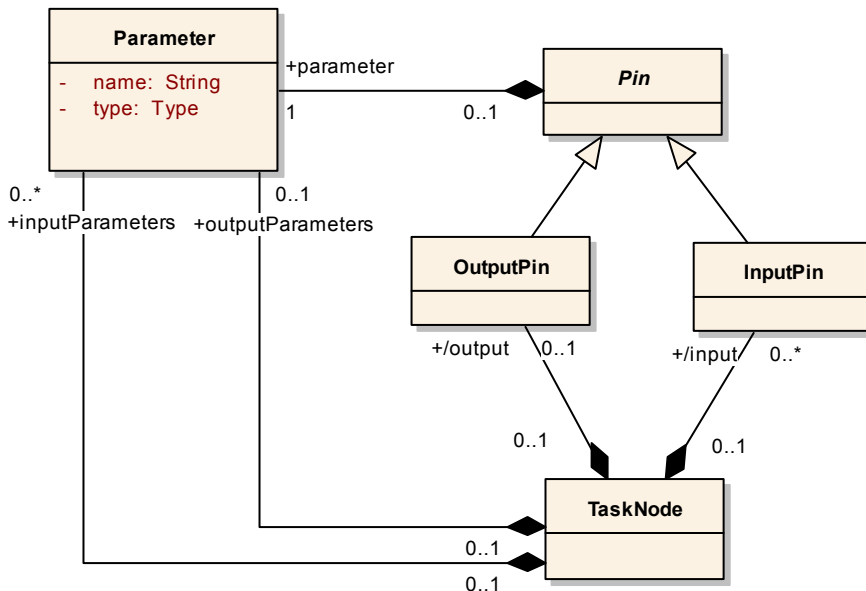


Figure 21 Object Flow constraint (metamodel)

Merges of output of several tasks to one input of a successor are done by defining a transformation, see section 5.7

Examples of different alternatives are presented below.

Data objects are used to represent the data content that is created in the composition and that may be passed along to different activities. Data objects may also represent the input and output parameters of the whole composition. There is a typical case in which one action results in an output object that is passed as input to the following action. In UML 1.5 one must model both the control flow and the object flow as two separate flows as in Figure 22. This case can be greatly simplified within UML 2.0.

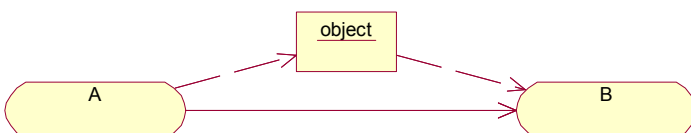


Figure 22 Object and control flow in UML 1.5

The control and object flow can be combined as shown in Figure 23. In UML 2.0 there is no difference between object and control flow. Both are modeled with an edge representing the flow.

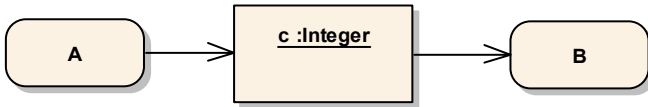


Figure 23 Object flow with implicit control flow in UML 2.0

There is a pin-notation Figure 24 in UML 2.0 that further may simplify the case above.

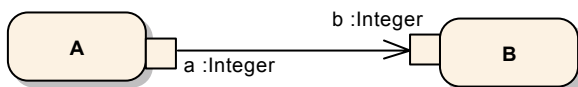


Figure 24 Pin notation for simplified object flow

Note that all the three figures above have equivalent semantics with different notation. One must be careful with using data objects in the composition in order to enable unambiguous interpretation when transformed into executable code. The main data object rules:

- A data object is produced once
- A data object is never updated. It will only be read by other services. An update must then be handled by some service reading the data object and producing a new data object
- A data object may be read by many services.

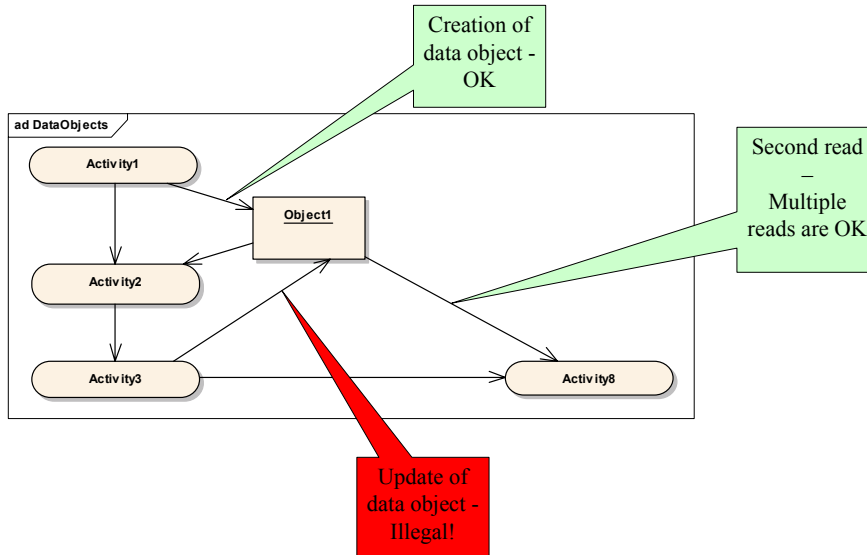


Figure 25 Updates of regular data objects are illegal

An exception to the main rule is allowed to handle list objects. The purpose of list objects is to ease the production of list structures in a composition without having to introduce many additional services. In order to avoid problems and ambiguity, the list object can only be used in a restricted manner. The list object is an ordered sequence of items, in which the items may also be lists themselves, if necessary. The list object may also handle collections of unordered items. In such a case, the consumers of the list structure will just ignore the order. A list object is created once by some service with a specialized *create* object flow, and the list will then be empty by convention. A list object may then be updated by specialized *add* object flows one or many times. We have introduced further restrictions because of the reading problem (Figure 26). *Activity7* in the diagram wants to read the list

data object. But when can we pass the list data object to the requesting activity? Objects that are never updated can be passed along to all requesters once it is produced, but list objects may be updated any time.

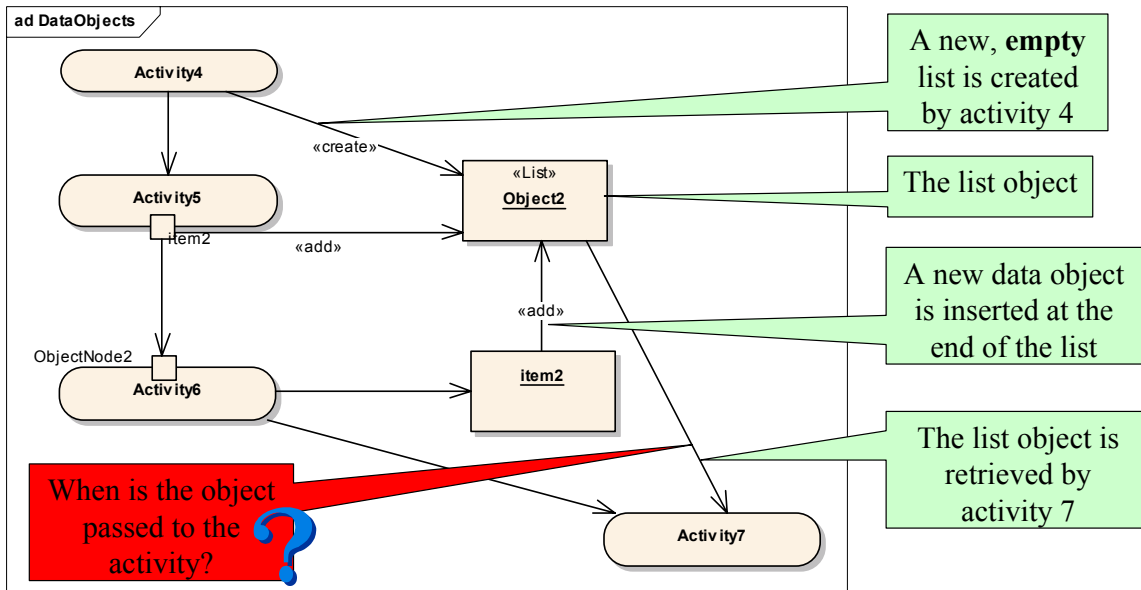


Figure 26 List objects: The reading problem

The restriction that solves the reading problem is to allow updates to a list object only inside a sub composition, and reading of the list object only outside the sub composition such as shown in Figure 27.

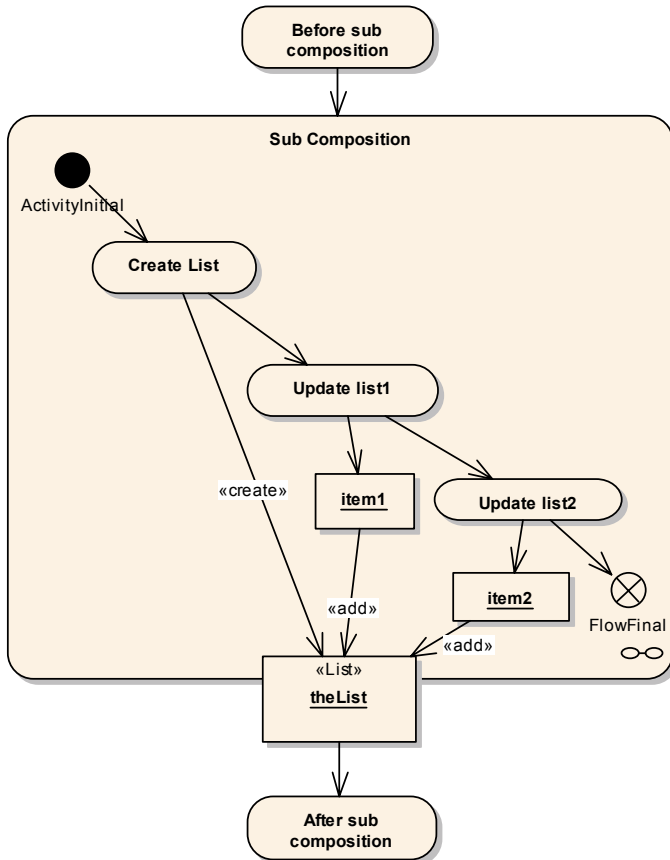


Figure 27 List objects: Updates inside sub composition, reading outside sub composition

5.2.2 Control flow

As previously mentioned, the ControlFlow represents the flow of control between TaskNodes directly or indirectly through ControlNodes (split or merge nodes).

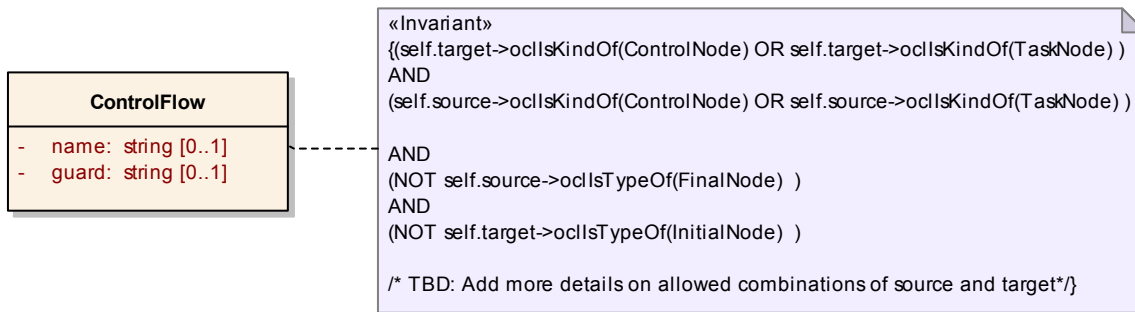


Figure 28 ControlFlow constraint (metamodel)

The rest of this section presents examples on how different kinds of control flow are modelled.

Sequence



Figure 29 Sequence (metamodel)

Figure 29 depicts flow of control (not data) between the two actions A and B. The arrow indicates the direction of the flow. This means that B executes its task when A has executed its task and passed on the control to B.

Exclusive OR

A decision node (diamond shape) is used to express that one of more possible paths/flows of control may be chosen among several choices, also known as a *Choice*.

Excluding guards are defined for each branch and the flow of control follows the path with a guard is *true*, also known as an XOR-split. Only one guard is true at the time. In Figure 30, the control from A will flow to either B or C, depending on the guard conditions specified on each branch following the decisionNode. It is good practice (www.modelingstyle.info/activityDiagram.html) to make the choices both exclusive and form a complete-set such that one and only one branch are followed after the decisionNode.

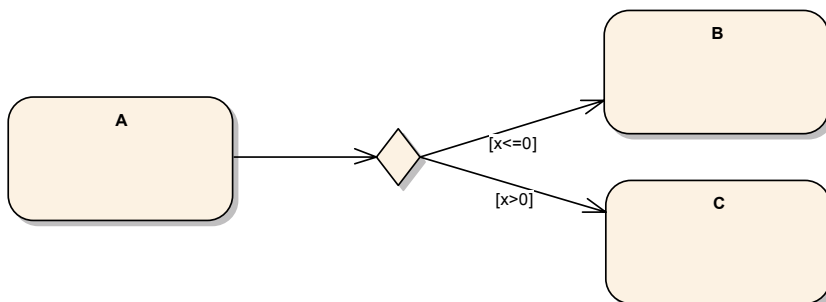


Figure 30 Exclusive OR (Choice, XOR-Split)

Merge

A merge is used to combine two or more flows into one flow. Its named *merge* since it will wait for the first token from its incoming flows and then continue to the outgoing flow. In Figure 31 the Merge will wait for a token from either action C or from action B, before the flow is continued with passing a token to the FlowFinal.

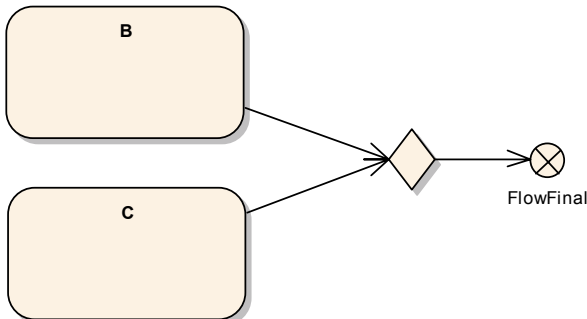


Figure 31 Merge (Or-Join)

Fork

A Fork is used to branch the flow from one incoming to many outgoing, concurrent flows. This is also called a And-split/Parallel split/Fork.

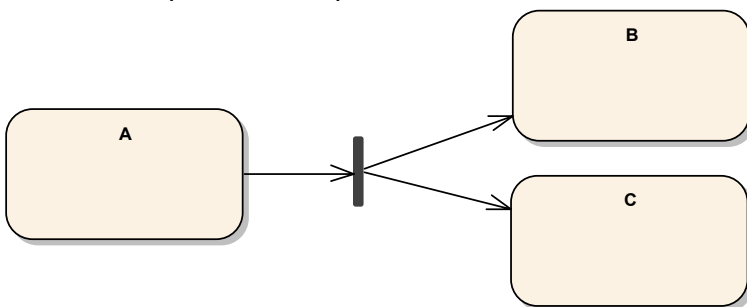


Figure 32 Fork (and split)

In Figure 32, the flow from A is split into two concurrent flows, which immediately will pass a token to the activities B and C.

Merge

A Merge is used to combine two or more flows into one flow, also known as *Synchronization* since the merge will wait for all incoming flows, before it passes a token to the outgoing flow. Also known as And-Join.

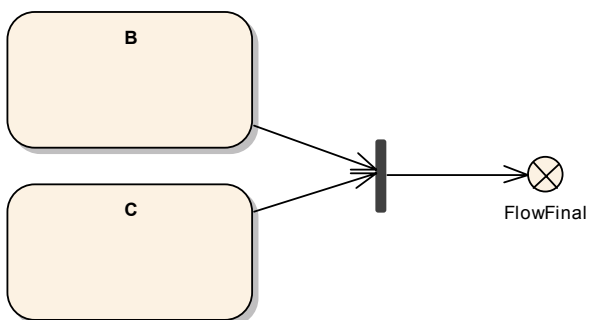


Figure 33 Merge

In Figure 33, the merge will wait for tokens from both action B and action C, before it passes a single token to the FlowFinal.

5.2.3 Event flow

EventFlow represents the flow of control from an *EventNode* to a *TaskNode*.

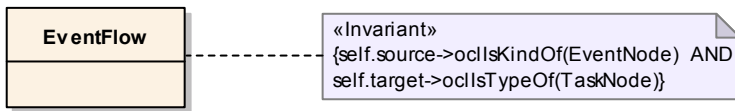


Figure 34 EventFlow constraint (metamodel)

Based on the platform user requirements (See SODIUM deliverables D3 and D5), the following two kinds of events might be interesting: *CancelEvent* and *TimeEvent*. In addition, a business process event (interrupt) was identified in D5- Part II, which occurs when the hospital personnel/doctor needs video clips from the accident location. *Events* are on the “may be supported” list so whether or not it will actually be implemented will be decided in the autumn of 2005 and will in that case be further explored in the next version of D9.

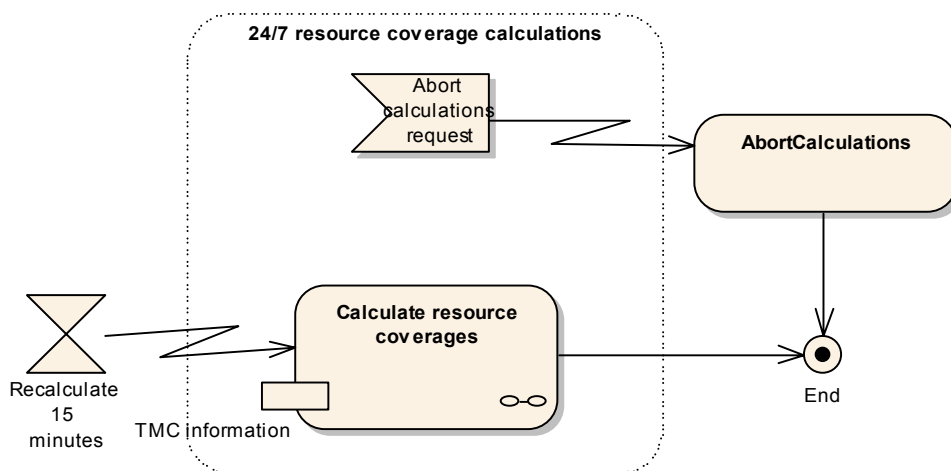


Figure 35 Time event and interrupt (example)

The two events in Figure 35 are Recalculate which occurs every 15 minutes (time glass shape) and the Abort calculation request. *Please note that this might as well also be implemented as part of the user application.*

5.3 Loop

This section presents the different kinds of loops that might be supported by the Visual Editor and VSCL2USCL translator. This further elaborated on and decided later on in the project. The next two subsections describe two kinds of loop-mechanisms defined in UML 2.0 that are used as input to this.

5.3.1 Loop Node

In UML 2.0 LoopNodes [22] have direct support for iterations as repeat-until, for-each etc. Currently example figures and notation information for the LoopNode is missing in the UML 2.0-draft. It is not stated what exact graphical notation to use other than that a LoopNode is a special kind of *Activity*. A LoopNode consists of setup, test and body sections. The setup section is executed once on entry to the loop, while the test and body sections are executed repeatedly for each loop. It has a boolean variable *isTestedFirst* that separates do-while/repeat-until from while-do. The three sections (setup,

test and body) all contain a set of nodes and edges. Until the notation and tool support for LoopNodes have been released this report suggests the notations below.

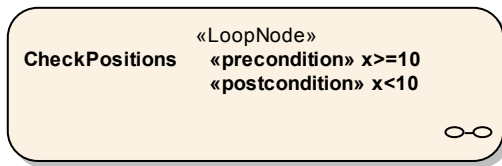


Figure 36 Loop example: Loop node (stereotyped UML subactivity)

Note: Loop nodes are on the “may be supported” list. Whether or not it will actually be implemented will be decided in the autumn of 2005 and will in that case be further explored in the next version of D9.

5.3.2 For each

The following example is based on a corresponding example in the requirements specification for the Crisis pilot D5 Part I.

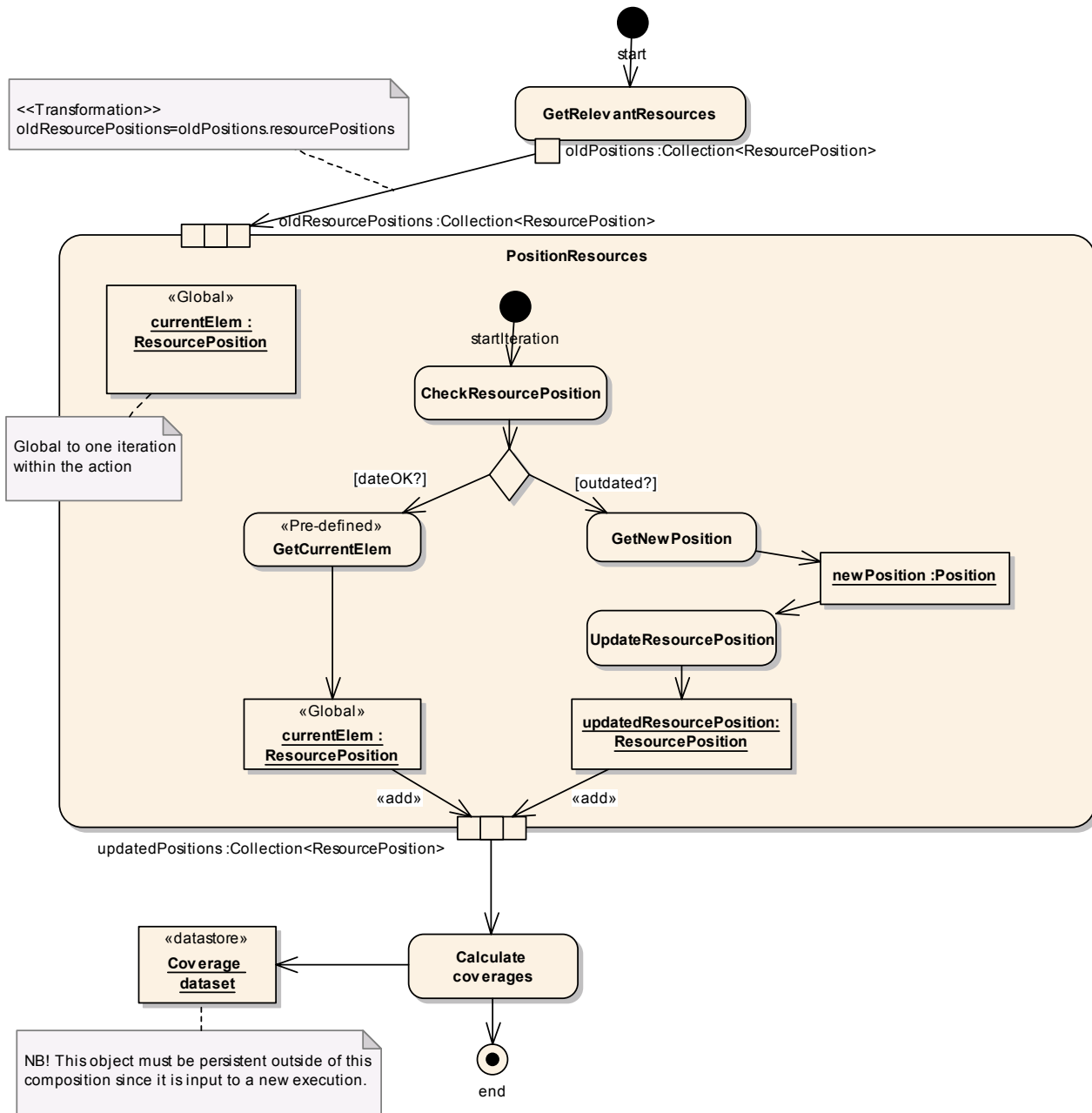


Figure 37 Loop example: For each (UML expansion region)

The data objects used in this model are presented in the figure below.

Note! Expansion region is on the “may be supported” list. Whether or not it will actually be implemented support for it will be decided in the autumn of 2005 and will in that case be further explored in the next version of D9.

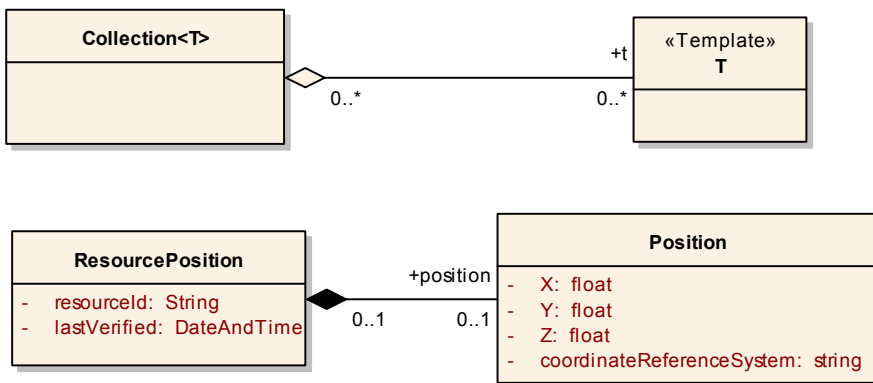


Figure 38 Loop- For each example (data model)

5.4 Discriminator

When working with services there is a great risk that a service is not responding. The server may have trouble, there may be too many simultaneous requests or the internet connection may be lost. Furthermore there may be a choice based on differences with respect to performance, quality of service and pricing. Due to this fact it is wise to have alternative services in the workflow model that perform the same task and we have a workflow model that is adaptable.

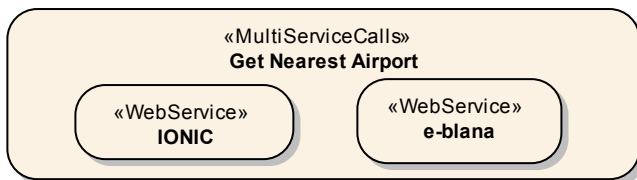


Figure 39 Alternatives as sub-actions

In this solution the task is modeled as an action of its own with a specified input data object and specified output data object. The rest of the workflow only relates to this information. The action is stereotyped as <<MultiServiceCalls>> to indicate that its task may be fulfilled by a set of alternative services. The alternative services are placed inside the MultiServiceCalls element as sub-actions Figure 39. The sub-actions may need data mappings to handle conversions between the input and output of the <<MultiServiceProviders>> element and the input and output of each sub-action. This is all modeled inside the <<MultiServiceCalls>> element. The sub-actions use the name of the provider as the name, since they all share the same logical name of the <<MultiServiceCalls>> element.

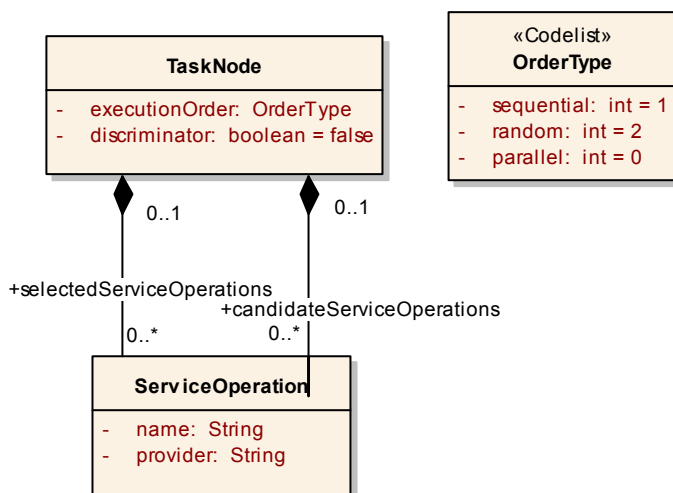


Figure 40 Discriminator (metamodel)

Figure 40 shows a relevant abstract from the metamodel which relates to the proposed language concepts for solving the discriminator pattern. A TaskNode may have a set of selected services that may execute the task. In addition it has two attributes: executionOrder and discriminator. The first is used for specifying in which order to execute the selected services (parallel, sequential or random). The order choice for sequential is handled by the visual editor. The discriminator is used to specify whether the execution should proceed to the next task when just one token is received (discriminator = *true*) or if all tokens should be received before proceeding (discriminator = *false*).

5.5 Conversations

The Conversations pattern is a communication form where there is interaction between a service requestor and a service provider where several messages are being sent back and forth. This is opposed to a single service call where the service requestor simply invokes a service providers service operation once and continues the flow towards other services. A conversation model is needed for scenarios which involves negotiation processes between requester and provider.

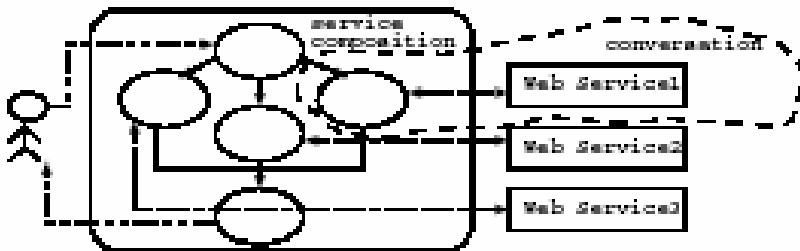


Figure 41 Service composition and conversation (taken from [28])

Figure 41 shows how Petrone [28] presents the relationship between service composition and conversation. A service composition involves the control flow between different tasks to be accomplished. Each of these tasks is realized by calls to a service. It may be a single call, but it may also be a conversation.

5.5.1 Example

A customer wants to buy some kind of goods. In order to get the best deal, the customer wants to request multiple suppliers for their offerings. By comparing the price offers, delivery method, product quality etc. the customer finally chooses to buy the goods from a chosen supplier.

The main actors in this scenario can be viewed as components: Customer, CompositeService, SupplierRegistry and Suppliers. The main component is the CompositeService which handles all the communication with the other actors. It can be viewed as a broker service that acts on behalf of the Customer by searching the SupplierRegistry for appropriate Suppliers, gathers price offers from the individual suppliers and notifies the suppliers of acceptance or rejection of the offer.

The example has been modelled without following any specific standard notation, but has combined notations from UML 2.0 Components diagram and Activity diagram (Figure 41). The notation is not part of the VSCL language yet, but illustrates the needs that will be addressed with a specific solution in VSCL later. Here is a chronological list of messages involved in the conversation:

- The Customer sends a Request For Proposal (RFP) to the CompositeService.
- The CompositeService sends a query to a supplier-registry (e.g. yellow pages) getting all appropriate suppliers in return.
- The CompositeService sends the RFP to all suppliers.
- The suppliers send their proposals to the service, asynchronously using a callback interface.
- The list of all proposals is sent to the customer.

- The customer decides which proposal to accept, and sends this information to the service.
- The service notifies all suppliers if their proposal was accepted (a) or not (b).
- The customer is given the details of the supplier behind the winning proposal.

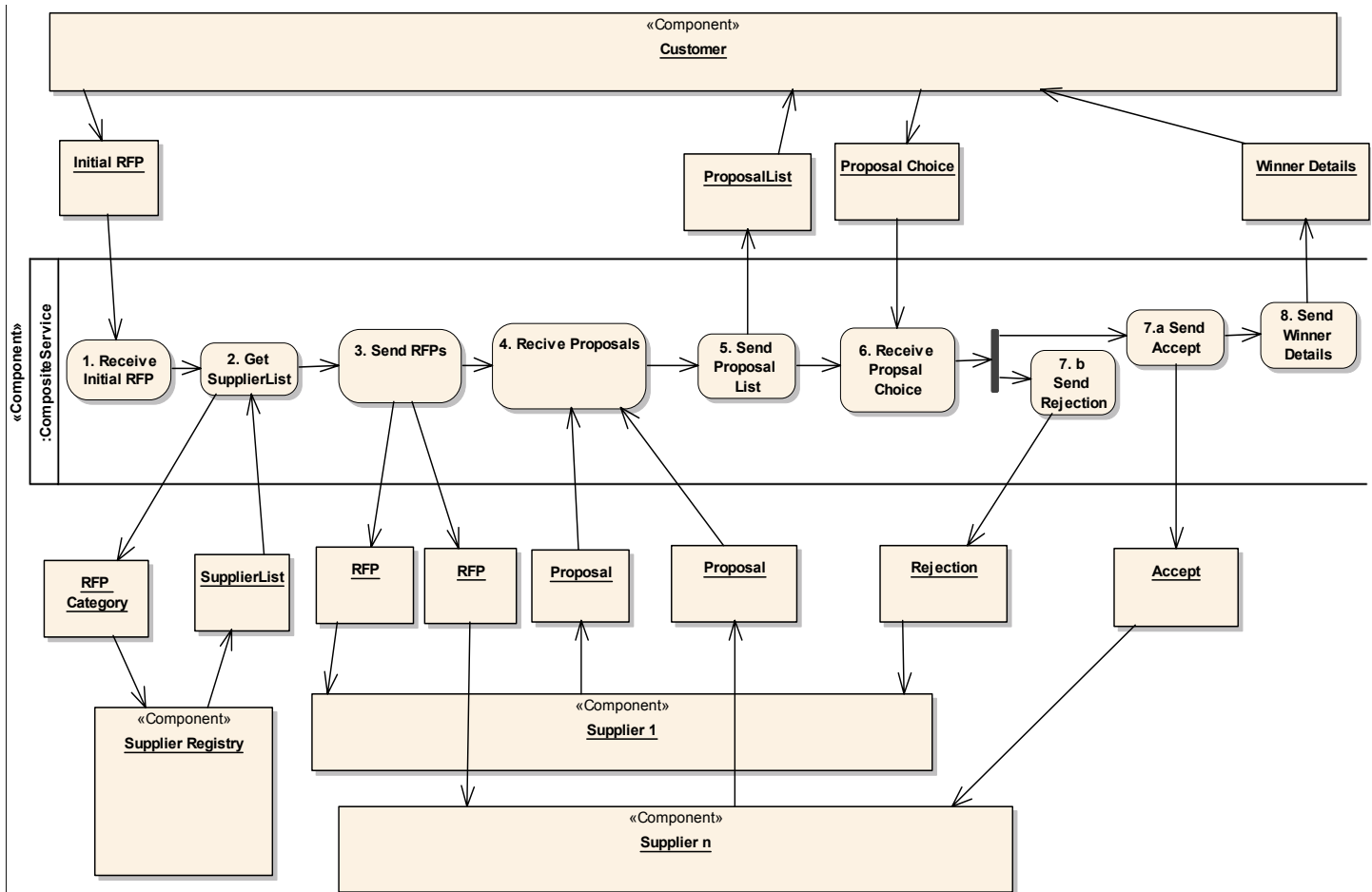


Figure 42 Conversation example (business process model)

Figure 43 shows the components involved in the interaction with their required interfaces and how they match with provided interfaces of other components.

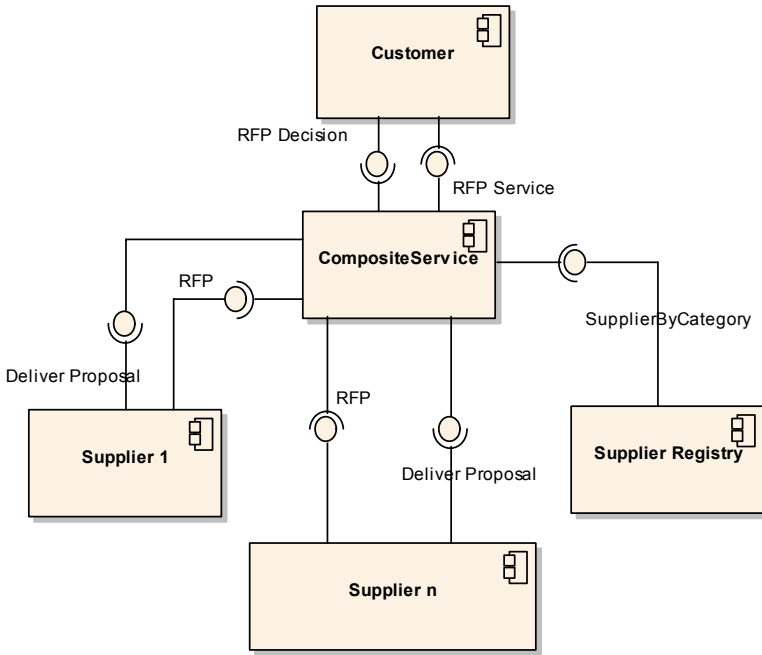


Figure 43 Conversation example (Component diagram)

5.6 Quality of Service (QoS)

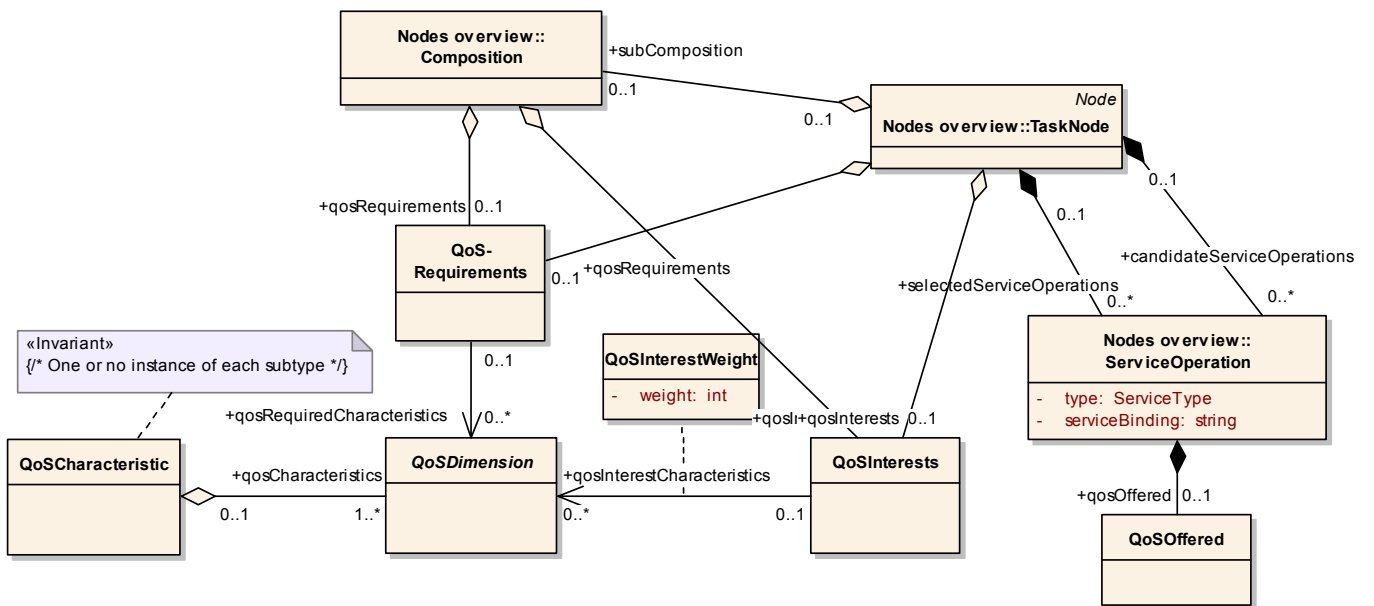


Figure 44 Quality Of Service

5.6.1 QoS Characteristics

The QoS concepts to be used needs to be precisely defined and used by all the parties involved. The OMG profile uses QoS characteristics to define collections of QoS concepts with precise semantic meaning. Each QoS characteristic contains a set of QoS dimensions with a name, its allowed value domain, if higher or lower values are considered better and its relationship to other QoS

characteristics. The direction of a characteristic is defined as either increasing or decreasing, where increasing means that higher values are preferred. All the QoS concepts to be used elsewhere shall be defined as a QoS characteristic either within the model itself or as in imported model. The modelling of QoS characteristics should be placed in a separate package or a separate model in order to simplify reuse.

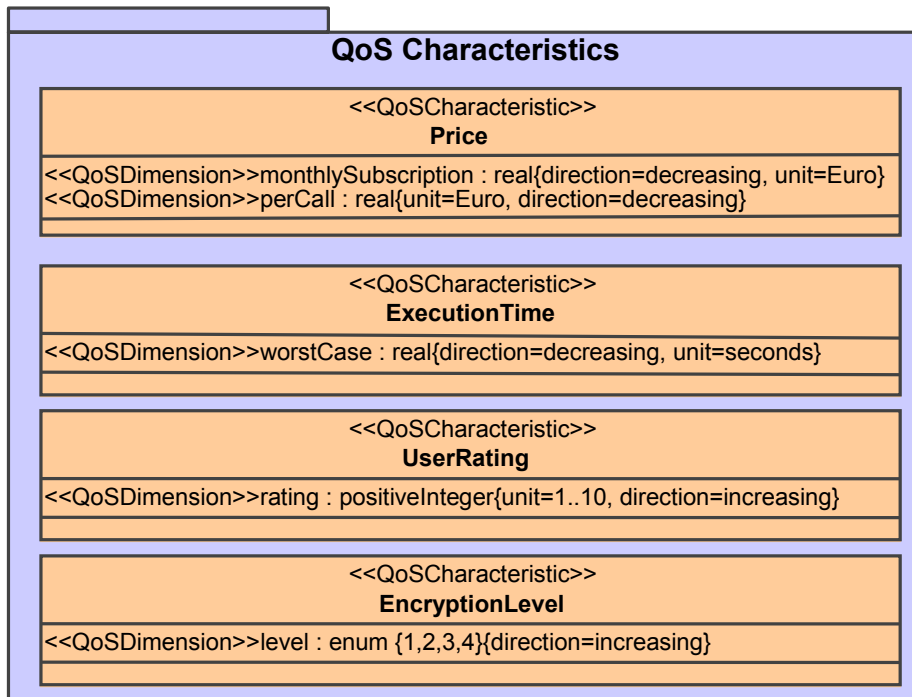


Figure 45 Modelling QoS characteristics

Note that VSCL by adopting the OMG profile only provides a generic means to define QoS ontologies. VSCL does not suggest any specific QoS dimensions as part of the language, but is capable of defining any needed QoS dimension. Figure 44 shows an example of five QoS dimensions which then serves as an example of how the VSCL language can be used to define a QoS ontology, and not as a fixed set to be supported by the SODIUM platform. The USQL language has a fixed set of QoS dimensions which should be modelled in VSCL when using the SODIUM framework. A price is either given as monthly subscription or per call and the currency is Euro. Execution time is measured in seconds and only with respect to worst case. A user rating specifies the user satisfaction on a scale from 1 to 10 where higher values are preferred. The encryption level ranges from 1 to 4 where higher values are preferred. These levels should be further detailed in order to be precisely defined.

5.6.2 QoS Requirements

After the QoS characteristics are defined, they can be used to model the QoS requirements. A QoS requirement identifies restrictions on a service with respect to the value domain of specific QoS characteristics. These requirements need to be fulfilled in order to achieve a successful binding to a service. The requirements may apply to a single service within the composition or the service composition as a whole. Note that the SODIUM platform does not support the handling of QoS requirements concerning the composition as a whole, since USQL QoS requirements can only be associated with single services. We also propose to extend the OMG profile with *QoS interests*, which is the set of QoS characteristics that are desirable to be optimized. The QoS interests follow the same principles as the other parts of the OMG profile such as the requirements modelling. It could be chosen that all the QoS characteristics referred to by the requirements are implicitly part of the Interest set. In many cases it may not be desirable since any characteristic that one tries to optimize may worsen the optimization of other characteristics as optimization deals with trade-offs. We suggest explicitly modelling all characteristics within the interests set. The result may then be that

characteristics specified only in the requirements set and not in the interests set will not be optimized any further beyond fulfilling the requirements upper and lower bounds. Furthermore we propose that the different characteristics in the interests set are weighted to indicate the relative importance of the interest. The SODIUM platform will not fully support this, since the USQL can only sort the result based on a single QoS dimension.

Figure 46 shows the workflow of the gas dispersion case extended with QoS requirements. The QoS requirements in the example (top left note) apply to the composition as a whole, since they are not attached to any the individual services (This is just a notational convention). The requirements state that the required price per invocation must be at most 50 Euros and the execution time must never exceed 20 seconds. In addition we specify QoS interests (lower left note) indicating which QoS characteristics we want optimized. Each of these interests has assigned a weight between 0 and 1, where 1 indicates highest importance. The QoS characteristics used are already defined by a separate UML package as shown in Figure 45.

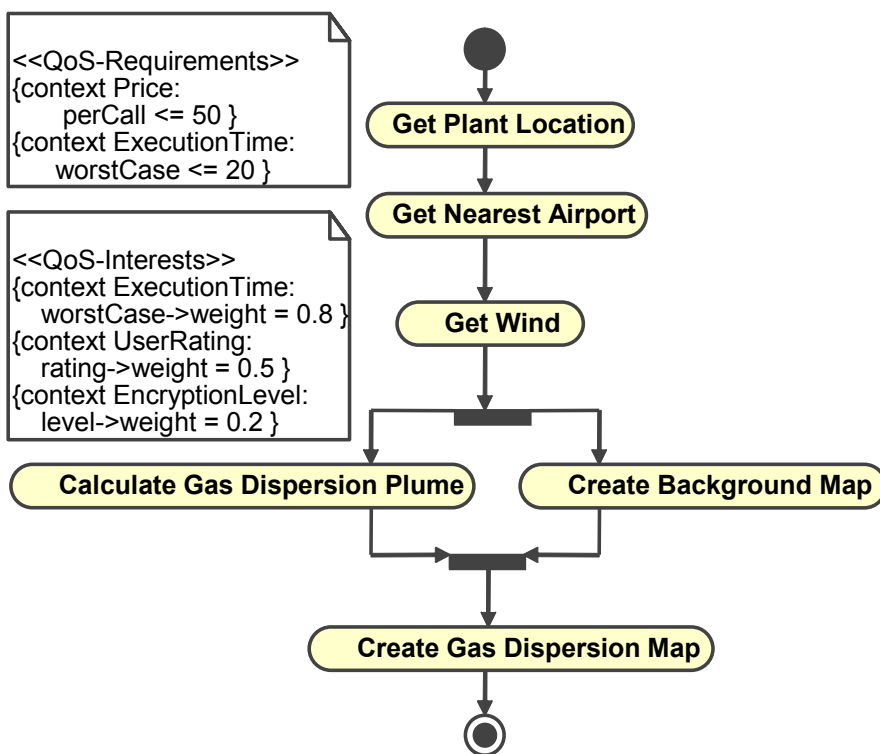


Figure 46 Modelling QoS requirements

5.6.3 QoS Offered

The composition model can show the selected services for each task including the individual offered QoS values of each service and the aggregated offered QoS for the composition as a whole. Figure 47 shows such a model for the gas dispersion case with imaginary offered QoS values. To simplify the figure we have not shown these WSDL details, only the name of the provider of the service. In UML we propose to specify the offered QoS values in a note attached to each task. The UML has also an unattached offered QoS note that by our convention indicates that it applies to the composition as a whole. The way of specifying the offered QoS value in the note follows the current draft OMG profile for QoS, while attaching them to the tasks and the activity model as a whole is a new way of applying the OMG profile. The current version of the OMG profile mentions only QoS offered for software components, classes or interfaces. The offered QoS values indicate the promised QoS for the individual services and the new Web service composition.

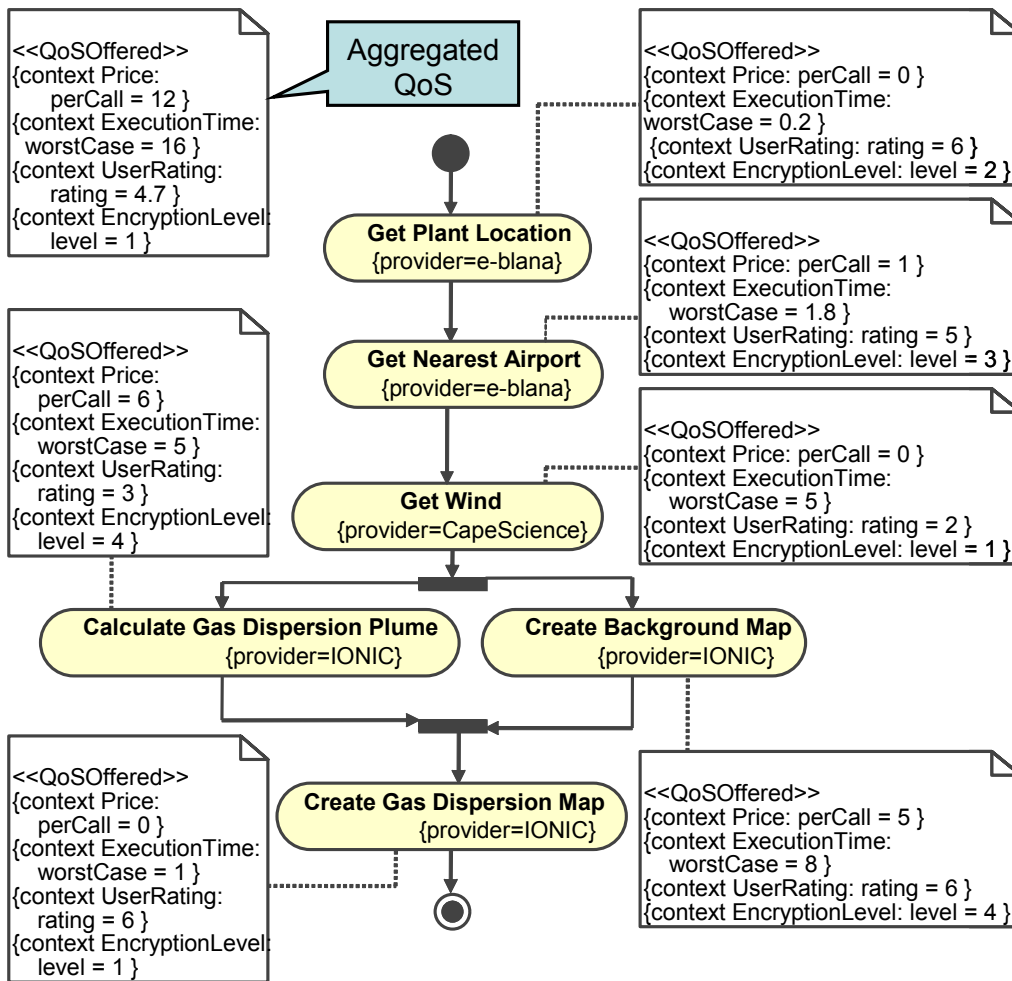


Figure 47 Composition model with actual QoS values

5.7 Transformation

When doing service composition, we are trying to assemble already existing services with fixed inputs and outputs. The output of one service does typically not exactly match the input of the next service. Therefore we need data transformations.

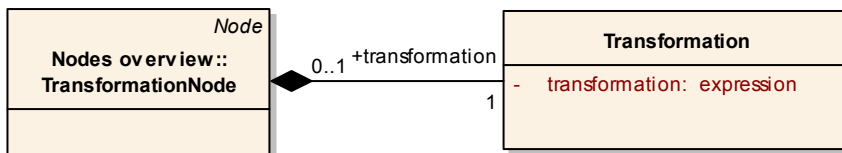


Figure 48 Data transformation (metamodel)

TransformationNodes are used for both simple one-to-one transformations and For more complex transformations (1:N, N:1 and N:M), see Figure 48. Whether the TransformationNode should be associated with the flows or pins will be elaborated further.

5.7.1 Discussion

There are different kinds of transformations. On one axis we may sort the transformations as belonging to exactly one of these categories:



- one-to-one,
- one-to-many,
- many-to-many, or
- many-to-one.

The cardinality refers to the number of data objects involved as source or target in the transformation. In a many-to-one transformation, the information from many source data objects is used to produce the content of a single target data object.

On another orthogonal axis we may sort the transformations as belonging to one or more of these categories:

- simple copying,
- renaming,
- assigning fixed values
- calculations and semantic mappings

Simple copying involves copying complete parts (not parts of a string, but part x1 of object x) of a source data object into parts of a target data object. Renaming means to use the same data object, but with a different name. Fixed values are hard-coded values to be inserted into parts of the target data object. Calculations and semantic mappings may involve conversions such as converting real values representing the speed, from km/h to m/s, translating a string value from lower case to upper case letters, adding together all the integer values contained in three different source data objects etc.

The left part of Figure 49 shows an example of an extract of a UML activity diagram that is about to be defined where a data transformation is needed compose two web services. The first web service, Get Ambulance Position, returns the position of an ambulance. After a call to the first web service, we want to call the next web service, Display Position On Map, which takes a position as input. At the logical level this seems rather easy as one would simply pass the output from the first web service on to the next web service. This is also how it typically works when all services are made from scratch within the same enterprise. But with existing services, outputs and inputs generally don't match. The middle part of Figure 49 shows the data types of the data objects shown in a UML class diagram. The right part of the figure shows a concrete example with actual data object instances represented as XML objects. Some kind of data transformation is obviously needed as these two data objects are different in shape although the content is the same. Only the bottom XML shape is accepted as input by the second web service.

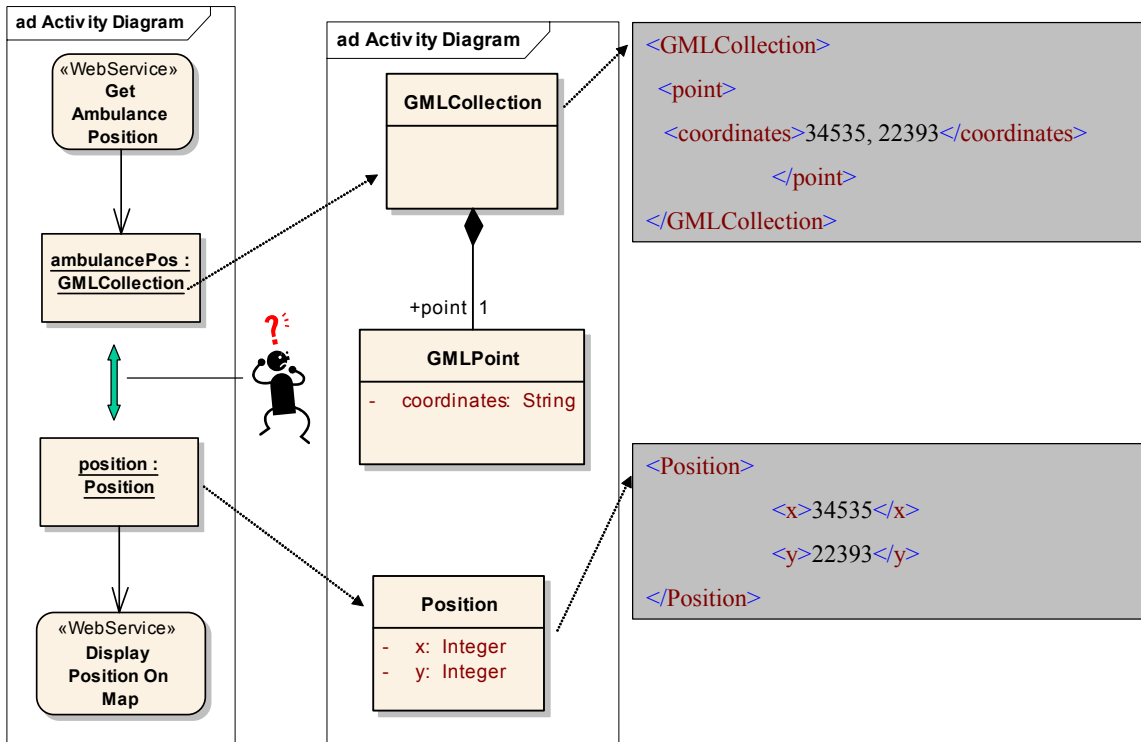


Figure 49 Data transformation needed

In order to make the VSCL language executable (meaning that it can be transformed automatically into something that can be executed), we need to represent this information inside the graphical environment. We present some alternatives in Figure 50. The left part shows how UML 2.0 suggests doing simple data transformations (*alternative 1*). Add a flow between the to-be-matched data objects. Add a <<Transformation>> stereotyped note to this new flow. Use some kind of transformation language inside this note. An alternative (*alternative 2*) is shown on the right part of the figure in which an intermediate activity is added which is stereotyped <<Transformation>> and has also a note attached defining the actual transformation. Thöne et al. [29] have a third approach quite similar to the right part where the transformation object is given a new symbol and where they propose to use XSLT expressions registered within a tagged value. A final alternative (*alternative 3*) would be to simply introduce a new service as an intermediate activity but without the transformation code. This would then be a completely new service that the service composition modeller needs to ensure is implemented.

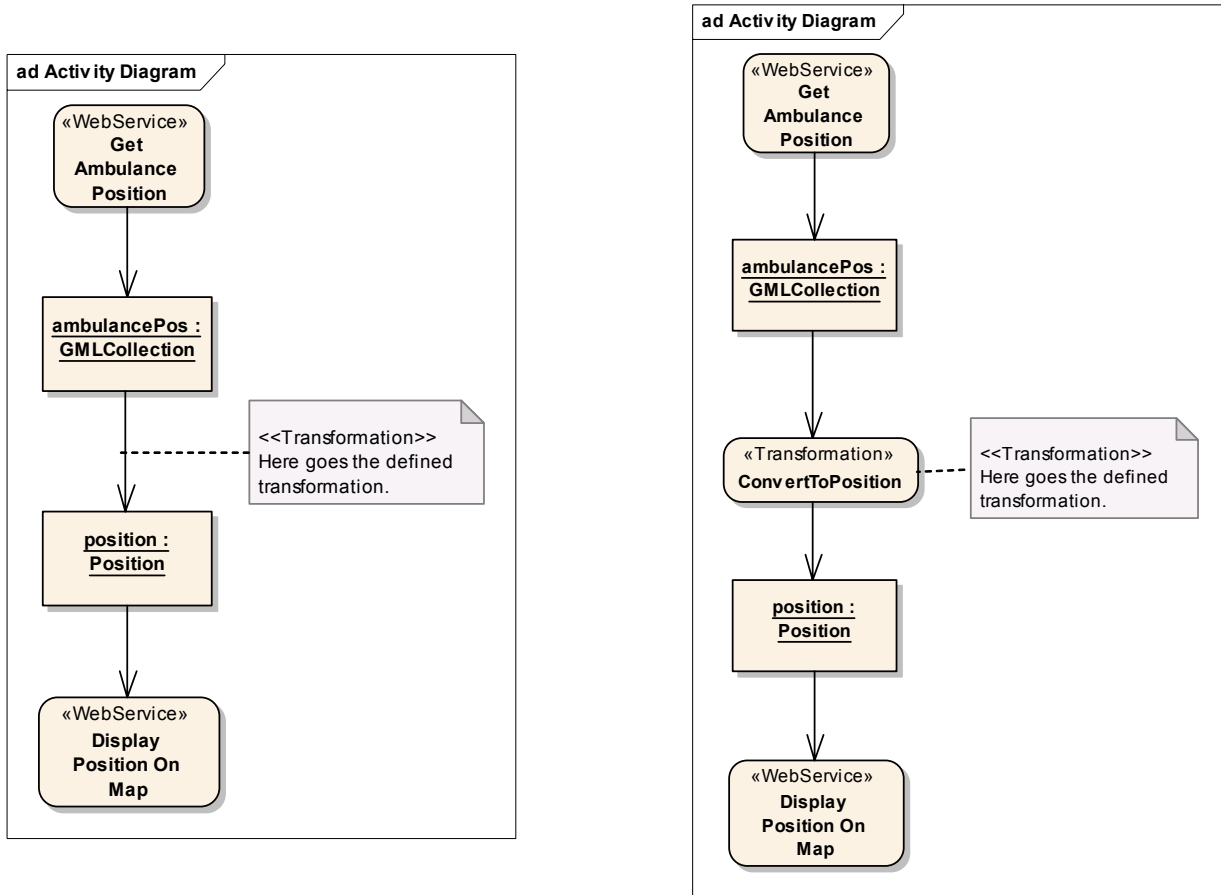


Figure 50 Alternatives for modelling data transformations

Advantages and disadvantages of the three main alternatives are described in the table below.

Table 1 Comparing data transformation alternatives

Alternative	Advantages	Disadvantages
Alternative 1 – transformation note attached to data flow	<ul style="list-style-type: none"> • Simple layout • Proposed in UML 2.0 	<ul style="list-style-type: none"> • Does not handle many-to-one transformations where the target result depends on input from more than one source data object • Hard to debug and ensure correctness of transformation note
Alternative 2- intermediate action with transformation note	<ul style="list-style-type: none"> • Handles all the different kinds of transformations 	<ul style="list-style-type: none"> • Clutters the diagram by more flow and additional activities (may be improved slightly by using a separate symbol as proposed by Thöne et al. [29]) • Hard to debug and ensure correctness of transformation note
Alternative 3 – new service to be implemented	<ul style="list-style-type: none"> • A transformation may be written in any chosen language with 	<ul style="list-style-type: none"> • The composition cannot be executed before the new service is implemented

Alternative	Advantages	Disadvantages
	transformation language tool support <ul style="list-style-type: none"> • Handles all the different kinds of transformations 	

We finally need to discuss what kind of transformation language to use inside the transformation stereotyped notes. XSLT or QVT [27] are probably the most promising choices. The advantage of XSLT is that it is specifically designed to do XML-to-XML transformations, and it is most likely that the data objects are defined as XML types by the service providers. With Web Services data objects are always of XML type. The same applies to Grid Services, but what about P2P Services?. A disadvantage at the model level is that the modeller needs to know details about XML in addition to the UML model platform. QVT is specifically designed to do model-to-model transformations and will be the natural choice if the actual data types also are modelled using UML. As many services will be published only as XML types, we need to have a reverse-engineering tool that automatically imports the XML types into UML types. Drawbacks with QVT are that the standard is not yet finalised, there are no available tools and the success of QVT is uncertain. XSLT on the other hand, is an already well-known and widely used standard. XQuery is a newer XML technology that can be an alternative to XSLT with many of the same advantages and disadvantages. The main difference is that XQuery has a non-XML notation that is inspired by the Structured Query Language (SQL). When using any of the alternative transformation languages inside the UML model, it will be challenging to validate concrete transformation definitions.

```

<<Transformation>>
<xsl:variable name="coord" select="/GMLCollection/point/coordinates"/>
<Position>
  <x>
    <xsl:value-of select=" substring-before($coord,',' )"/>
  </x>
  <y>
    <xsl:value-of select=" substring-after($coord,',' )"/>
  </y>
</Position>

```

```

<<Transformation>>
- Assumes the existence of a this library function:
- indexOf [in String] : Integer {}

mapping ambulancePos2position [in GMLCollection] :
Position {
  var coord := self.point.coordinates;
  var commaPos := indexOf(coord);
  x := coord := substring( 1, commaPos - 1);
  y := coord := substring( commaPos + 1, coord.size() );
}

```

Figure 51 XSLT vs. QVT

5.7.2 Suggested approach and major challenges

We suggest mixing all the three different alternatives depending on the problem at hand. We believe that the most typical case are simple transformations only involving one-to-one transformations with simple copying. Therefore we propose to use the alternative 1 (transformation notes attached to the object flow) as the default transformation notation in SODIUM. Since it does not handle more complex transformations we propose to use alternative 2 (intermediate transformation activities) in those cases. For the most complex transformations it may also be most suited to define a completely new service without the data transformation code defined inside the model. We recommend to use XSLT as the transformation language in SODIUM since this requires less effort and is also supported by the USCL language, and thus by the SODIUM platform.

In the future and outside of the SODIUM project, we plan to experiment with using QVT as the transformation language. Hopefully some UML tools in the future will implement QVT language support so that the transformation expressions may be validated. There are a few major challenges that need to be investigated:

- How to validate that the QVT data transformations are valid

- How to ensure that all the transformations work properly and that a correct, unique execution result is finally implemented. How do we translate from XML types into UML and back again from QVT expressions to XSLT when running this in practice? Can we be sure that it will work? If we mix an XML attribute name with an XML element name things will not work.

5.8 Semantics

An ontology concept is called `OntClass`. The `OntClass` is a specialization of every type and the `OntClasses` are grouped inside an `Ontology` package. The `OntClass` is equivalent to an `owl:Class` in the OWL Web Ontology Language (OWL) [30]. This follows from the UML Ontology Profile (UOP) defined by Djuric [31] who have defined a mapping between UML and OWL. We have not detailed the `OntClass` any further than having a name which is unique within the `Ontology` package. The `Ontology` package is identified by a URI. We do not intend to model the ontology concepts any further than referring to concepts defined elsewhere. If there is a need to also see the details of an ontology concept, then we propose to use the OWL to UML mapping defined by Djuric. Although the complete OWL to UML mapping is part of the VSCL language, we may choose to implement a subset of this in VSCL editor. The subset should be large enough to support the needs of the SODIUM pilots.

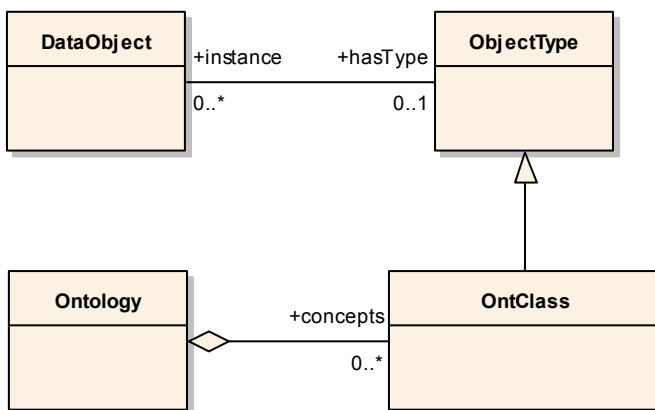


Figure 52 Semantic information (metamodel)

The VSCL language shall support semantics in order to fully generate USQL documents with such information. It may also be relevant to be able to reverse-engineer existing descriptions in the leading semantic web specifications OWL-S and WSMO. The VSCL language will support USQL 100% and a subset of OWL-S and WSMO. VSCL shall not be used to define new ontologies, but VSCL will be used to define all the data types by references to existing ontology concepts. Four major semantic properties may be associated with a service: input, output, preconditions and effects (IOPE). Logic expressions may be used to express preconditions and effects, while input and output are typically references to the proper ontology concepts. VSCL will support the input and output part, but postpones the preconditions and effects. Every data object used as input and output within data objects should be defined with a type represented by a class which is stereotyped as `<<OntClass>>`. The `<<OntClass>>` is equivalent to an `owl:Class`. This follows from the UML Ontology Profile (UOP) defined by Djuric [31] who have defined a mapping between UML and OWL. The ontology concept will then be placed inside an `<<Ontology>>` stereotyped UML package showing which ontology the ontology concept belongs to. An example is shown in the figure below. The `CreateGasDispersionMap` activity has two input parameters and one output parameter. All these three parameters need to be linked to ontology concepts. This is achieved by letting all the three parameter types be ontology concepts themselves.

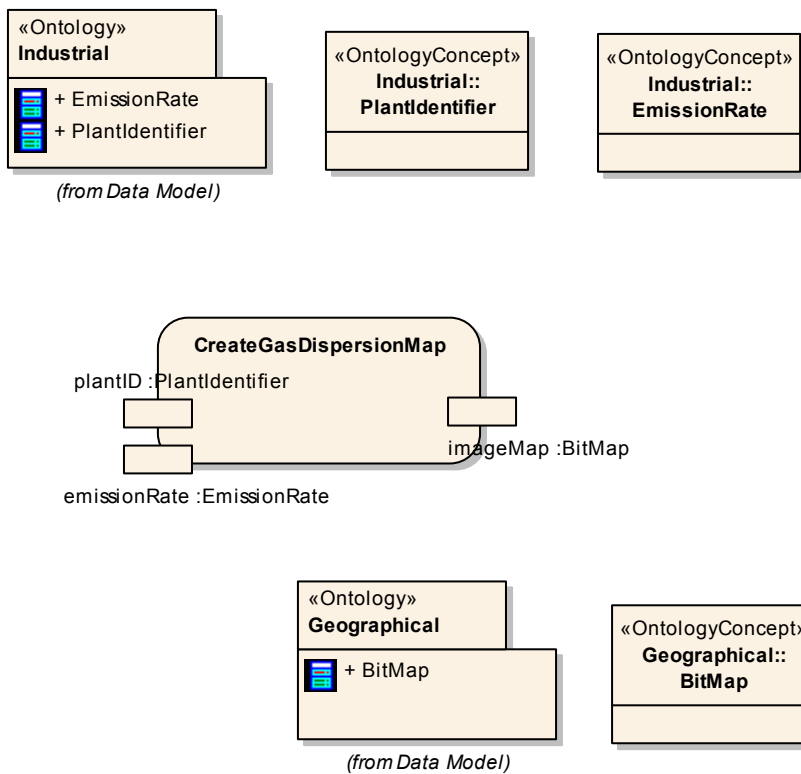


Figure 53 Identifying ontology concepts for the inputs and outputs of a service

We have also manually reverse engineered the OWL-S CongoExpress example from <http://www.daml.org/services/owl-s/1.0/examples.html> which shows another example of modelling with semantic references Figure 53. There is one service which provides a single operation `expressCongoBuy` (`ExpressCongoBuyInput`): `ExpressCongoBuyOutput`. Both the input and output data objects are defined as classes which themselves contain of attributes with attribute types being ontology concepts.

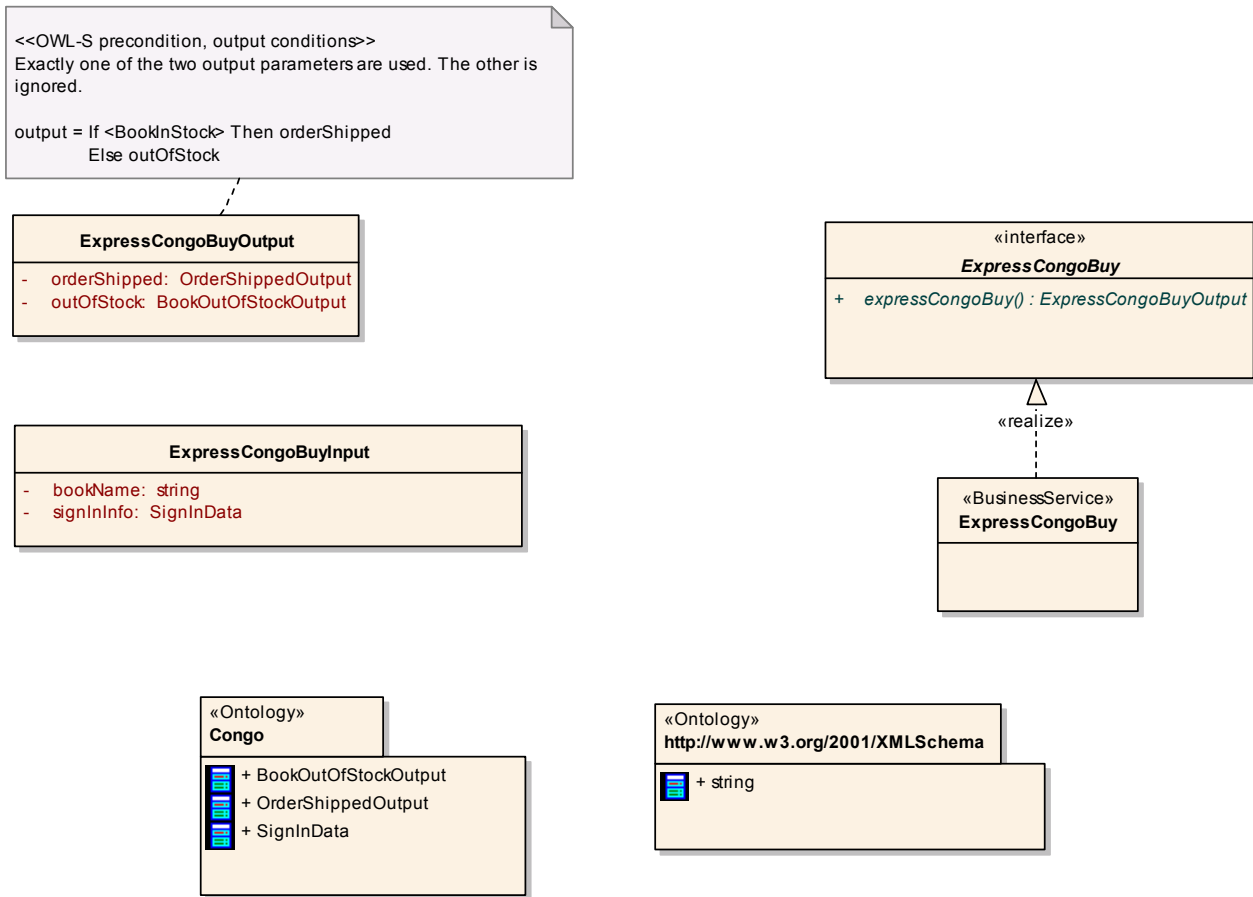


Figure 54 The OWL-S CongoExpress example modelled in UML

6. Static structure

This chapter presents the metamodel for static modelling in VSCL. This includes:

- the modelling of the web service interface for those service compositions to be
- a component diagram that presents the exposed and required interfaces of the different services involved in the composition.

6.1 Web service interface for the composition

The resulting composition may itself be exposed as a service. SODIUM has decided that only generation of Web Service descriptions will be supported by the prototype. For the new service its operation needs to be defined with i.e. operation name, input and output parameters.

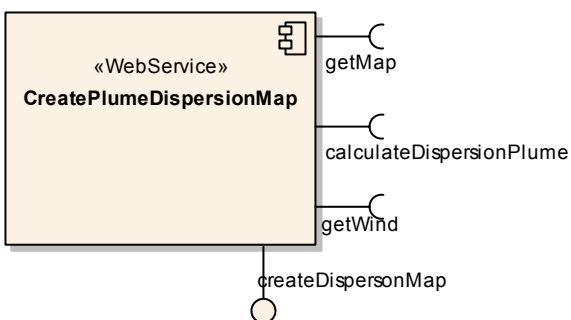


Figure 55 Service interface example

Figure 55 shows a Web service with one provided interface and three required interfaces. The interior of the Web Service is not exposed, but it could be a service composition. The possibility of defining the interface of a composition will be elaborated further in deliverable D9 and will be further investigated in deliverable D12 that both concern the SODIUM Visual Service composition Suite.

6.2 Component diagram presenting for the interface relationships of the composition services

The model in Figure 56 shows an alternative view on the composition. This view focuses on which services are selected for each of the required interfaces, but the composition flow of data and control is not shown.

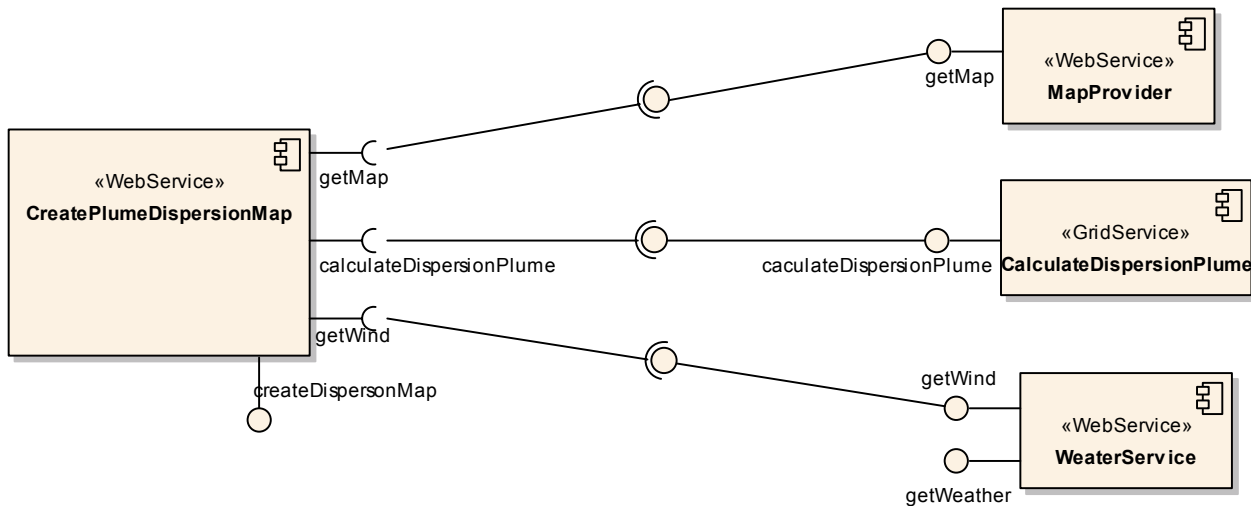


Figure 56 Interface relationship example

This view is not going to be implemented in the SODIUM Visual Editor prototype, but is identified as a useful, future extension to the tool. Other possible extensions may also be identified and investigated further in the next project phase as part of deliverable D9.



7. Conclusions and further work

This document has presented the concepts of the SODIUM Visual Composition Language (VSCL). Most of the concepts will be supported by the SODIUM Visual Service Composition Suite (deliverable D9) for *heterogeneous service composition*. The suite consists of three main parts: the Visual Editor where the concepts of VSCL will be made available, the VSCL2USCL and VSCL2USQL translators, and the Service Composition Analyzer.

The VSCL has a conceptual meta-model that is independent of UML and other existing graphical modeling languages. The main concepts of the languages are the *tasks* and the *flow of data and control* between tasks. A task consists of both an abstract part and a concrete part. It may be coarse-grained which means that it can be detailed in a sub composition of tasks. The abstract part is service independent and may be used as a starting point when querying for available and relevant heterogeneous services. The concrete part of the task has information about which service(s) to execute for the specific task. The strength of the task approach is that there is one composition graph, and not two – one concrete and one abstract. After services have been selected, the abstract part may still be used to check if new available and better suited services have emerged. To match task output(s) to the input(s) of the successor task, the possibility for defining transformations have been included.

VSCL also contains concepts for defining *Quality of Service (QoS)* requirements for the tasks³ and add *Semantics* to them by annotating various properties of each task (service category, and the task' input and output parameters). These are used as criteria for service discovery, sorting and selection and are very important mechanisms for improving the effectiveness and the precision of the service discovery process supported by the Unified Service Query Engine prototype. This engine will support both query for ontology concepts to be used for semantic annotation and query for heterogeneous services in a unified way.

The heterogeneous service compositions will not be executed directly, but are first translated into the lexical Unified Service Composition Language (USCL) and then executed by the SODIUM Execution Engine.

The ideas of VSCL have been input to the specification of the SODIUM Visual Service Composition Suite. The next step is to implement this suite and let it be validated and tested by the users. The experiences gained during the implementation and the testing/validation will be valuable input to the next version of VSCL and its supporting tools. The ideas and experiences will be documented and published at various arenas and will also serve as input to ongoing standardization work.

³ Note that tasks are independent of service type (p2p, grid or web service).



8. References

- [1] Skogan, D., R. Grønmo, and I. Solheim. *Web Service Composition in UML*. in *The 8th International IEEE Enterprise Distributed Object Computing Conference*. 2004. Monterey, California.
- [2] Grønmo, R. and I. Solheim. *Towards Modeling Web Service Composition in UML*. in *The 2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI-2004)*. 2004. Porto, Portugal.
- [3] OASIS, *Web Services Business Process Execution Language (WS BPEL)*. 2004, <http://www.oasis-open.org/>.
- [4] Woodman, S.J., et al. *Notations for the Specification and Verification of Composite Web Services*. in *The 8th International IEEE Enterprise Distributed Object Computing Conference*. 2004. Monterey, California, USA.
- [5] BPMN, *Business Process Modeling Notation (BPMN)*. 2003, Business Process Management Initiative (BPMI), <http://www.bpml.org/>.
- [6] White, S.A., *Process Modeling Notations and Workflow Patterns*, in *The Workflow Handbook 2004*, L. Fischer, Editor. 2004, Future Strategies Inc.
- [7] OMG, *Business Process Definition Metamodel - Revised submission to OMG BEI RFP bei/2003-01-06*. 2004
- [8] OMG, *UML for EDOC*, www.omg.org/techprocess/meetings/schedule/UML_Profile_for_EDOC_RFP.html.
- [9] OMG, *UML for EAI - UML Profile for Event-based Architectures in Enterprise Application Integration (EAI) - final adopted specification*. 2002
- [10] Pautasso, C. and G. Alonso, *JOpera: a Toolkit for Efficient Visual Composition of Web Services*. 2003
- [11] Coalition, T.O.S., *OWL-S: Semantic Markup for Web Services*. 2004
- [12] WSMO, *Web Service Modeling Language*. 2005, <http://www.wsmo.org/wsml/>.
- [13] Krishnan, S., P. Wagstrom, and G.v. Laszewski, *GSFL: A Workflow Framework for Grid services*. 2002.
- [14] OMG, *MDA Guide Version 1.0.1*, J.M.a.J. Mukerji, Editor. 2003, OMG
- [15] W3C, *Web Services Architecture W3C Working Group Note*. 2004, <http://www.w3c.org/TR/ws-arch/>.
- [16] INTEROP, W., *State-of-the art for Interoperability architecture approaches*. 2004, Network of Excellence - Contract no.: IST-508 011, www.interop-noe.org.
- [17] P. Chatterjee, J.W., *Developing Enterprise Web Services and Applications: An Architect's Guide*. Hewlett-Packard Professional Books. 2003: Prentice Hall Professional Technical Reference, ISBN: 0131401602.
- [18] Aalst, W.M.P.v.d., et al., *Workflow Patterns*. 2000, BETA Working Paper Series, Eindhoven University of Technology: Eindhoven. p. 1-70, <http://tmitwww.tm.tue.nl/research/patterns/documentation.htm>.
- [19] Aalst, W.M.P.v.d., et al., *Workflow Patterns*. Distributed and Parallel Databases, 2003. **14**(3): p. 5-51.
- [20] Aalst, W.M.P.v.d., *Don't go with the flow: Web Services composition standards exposed*, in *IEEE Intelligent Systems*. 2003. p. 72-76
- [21] Aalst, W.M.P.v.d., et al. *Advanced Workflow Patterns*. in *7th International Conference on Cooperative Information Systems (CoopIS 2000)*. 2000: Springer-Verlag, Berlin.
- [22] OMG, *UML Infrastructure 2.0 Draft Adopted Specification*. 2003.



-
- [23] Rumbaugh, J., Jacobson, I., Booch, G., *The Unified Modeling Language Reference Manual*. Second ed. 2004: Addison-Wesley, ISBN: 0-321-24562-8.
- [24] Fowler, M., *UML Distilled*. Third ed. 2004, Boston: Addison-Wesley, ISBN: 0-321-19368-7.
- [25] OMG, *Unified Modelling Language (UML)*. 2004, Object Management Group, <http://www.uml.org>.
- [26] Warmer, J., Kleppe, A., *The Object Constraint Language - Getting your models ready for MDA*. Second ed. 2003: Addison-Wesley, ISBN: 03211179366.
- [27] OMG, *Object Management Group MOF 2.0 Query / Views / Transformations RFP*. 2002, www.omg.org.
- [28] Petrone, G. *Managing flexible interaction with Web Services*. in *Workshop on Web Services and Agent-based Engineering in conjunction with the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'2003)*. 2003. Melbourne, Australia.
- [29] Thöne, S., R. Depke, and G. Engels. *Process-Oriented, Flexible Composition of Web Services with UML*. in *Int. Workshop on Conceptual Modeling Approaches for e-Business: A Web Service Perspective (eCOMO 2002)*. 2002. Tampere, Finland.
- [30] W3C, *OWL Web Ontology Language - W3C Recommendation 10 February 2004*. 2004, <http://www.w3.org/TR/owl-features/>.
- [31] Djuric, D., *MDA-based Ontology Infrastructure*. *Computer Science and Information Systems (ComSIS)*, 2004. **1**(1): p. 91-116.
- [32] Pillana, S., et al., *Towards an UML Based Graphical Representation of Grid Workflow Applications*. 2004.
- [33] Cybok, D., *A Grid Workflow Infrastructure*. presented at GGF10 - Grid Workflow Workshop, Berlin, Germany, 2004.
- [34] Hoheisel, A. *User Tools and Languages for Graph-based Grid Workflows*. in *GGF10 – Grid Workflow Workshop – User Tools & Languages*. 2004.
- [35] Dept. of Computer Science, R.U., USA, *Will Reliability Kill the Web Service Composition?*
- [36] Majithia, S., Walker, D. W., Gray, W. A., *Automated Composition of Semantic Grid Services*,. presented at UK e-Science All Hands Meeting, Nottingham, UK, 2004.
- [37] M. Papazoglou, B.K., J. Yang, *Leveraging Web-Services and Peer-to-Peer Networks*. *Caise 2003, LNCS 2681*, 2003: p. 485-501.



9. Annex A – Requirements and patterns

Workflow patterns

The patterns listed in the subsequent table and requirements are further described in [18-21].

#	Pattern	Source	To be supported
Basic control flow patterns			
1	Sequence	[18-21]	x
2	Parallel split	[18-20]	x
3	Synchronization	[18-20]	x
4	Exclusive Choice	[18-20]	x
5	Simple Merge	[18-20]	x
Advanced branching and synchronization patterns			
6	Multi-choice	[18-20]	x
7	Synchronizing merge	[18-20]	x
8	Multi-merge	[18-20]	x
9	Discriminator	[18-20]	x
10	<i>N</i> -out of <i>M</i> - Join	[18]	
Structural patterns			
11	Arbitrary cycles	[18-20]	
12	Implicit termination	[18-21]	
Patterns involving Multiple Instances			
13	Multiple Instances With a Priori Design Time Knowledge	[18-20]	
14	Multiple Instances With a Priori Runtime Knowledge	[18-20]	
15	Multiple Instances With No a Priori Runtime Knowledge	[18]	
	Multiple Instances Without a Priori Runtime Knowledge	[19, 20]	
16	Multiple Instances Requiring Synchronization	[18, 21]	
	Multiple Instances without synchronisation	[19, 20]	
Temporal relations			
17	Interleaved sequence	[18]	
State-based Patterns			
18	Deferred XOR-split	[18]	
	Deferred Choice	[19-21],	
19	Interleaved parallel routing	[18-20]	
20	Milestone	[18-20]	
Cancellation patterns			
21	Cancel activity	[18-20]	
22	Cancel case	[18-20]	
Inter-Workflow Synchronization			
23	Messaging communication	[18]	
24	Messaging coordination	[18]	
25	Bulk message sending	[18]	
26	Bulk message receiving	[18]	

The following table is captured from [20].

Table 1: Comparison of BPEL4WS, XLANG, WSFL, XPDL, and four workflow products.

	BPEL4WS	XLANG	WSFL	XPDL	Staffware	MQ Series Workflow	Panagon eProcess	FLOWer
Pattern 1 (Sequence)	+	+	+	+	+	+	+	+
Pattern 2 (Parallel Split)	+	+	+	+	+	+	+	+
Pattern 3 (Synchronization)	+	+	+	+	+	+	+	+
Pattern 4 (Exclusive Choice)	+	+	+	+	+	+	+	+
Pattern 5 (Simple Merge)	+	+	+	+	+	+	+	+
Pattern 6 (Multi-choice)	+	-	+	+	-	+	+	-
Pattern 7 (Synchronizing Merge)	+	-	+	-	-	+	+	-
Pattern 8 (Multi-merge)	-	-	-	-	-	-	-	+/-
Pattern 9 (Discriminator)	-	-	-	-	-	-	-	+/-
Pattern 10 (Arbitrary Cycles)	-	-	-	+	+	-	+/-	-
Pattern 11 (Implicit Termination)	+	-	+	+	+	+	+	-
Pattern 12 (Multiple Instances Without Synchronization)	+	+	+	-	-	-	+	+
Pattern 13 (Multiple Instances With a Priori Design Time Knowledge)	+	+	+	+	+	+	+	+
Pattern 14 (Multiple Instances With a Priori Runtime Knowledge)	-	-	-	-	-	-	-	+
Pattern 15 (Multiple Instances Without a Priori Runtime Knowledge)	-	-	-	-	-	-	-	+
Pattern 16 (Deferred Choice)	+	+	-	-	-	-	-	+/-
Pattern 17 (Interleaved Parallel Routing)	+/-	-	-	-	-	-	-	+/-
Pattern 18 (Milestone)	-	-	-	-	-	-	-	+/-
Pattern 19 (Cancel Activity)	+	+	+	-	+	-	-	+/-
Pattern 20 (Cancel Case)	+	+	+	-	-	-	+	+/-

Requirements from literature

A review of relevant research papers and reports are used to gather a list of identified requirements to grid, P2P and web service composition. These requirements may be at different levels of detail and relevant in different contexts. This list does not sort these requirements in any way or filter them so that they necessarily are relevant for SODIUM. This filtering process is the next step in SODIUM. A thorough reason should be given for why published requirements are not relevant for SODIUM.

For each discovered requirement we give these details:



- *Name*. Short name or sentence of the requirement.
- *Definition*. Precise description of the requirement
- *Relevance*. If it is identified for grid, P2P or web services and if this is a requirement that the authors state are beyond the requirements of any of the three paradigms. (For instance a requirement may be identified as a grid requirement that goes beyond requirements in web services as opposed to a grid requirement that is also relevant for web services)
- *Implications*. Does this have special impact on the architecture of the composition or the execution environment?
- *Literature*. The list of research papers or reports that identify the requirement. It should be clarified in this section if there are differences with respect to naming, definition or relevance to grid, P2P or web services as stated by the authors.

Requirement : Large data amounts

Definition: The size of the data objects to be transferred from one service to another can be very large.

Relevance: Grid. Goes beyond web services

Implications: This means that a central workflow engine should not be used for exchanging the data as it can be a bottleneck for exchanging large amounts data. Instead only references to the data should be used or they should be exchanged in a peer-to-peer manner between the services instead of involving a central coordination point. [13] proposes to use asynchronous messages as notificationSources and notificationSinks which eliminates the need to use the central coordination point.

Literature: [13, 32, 33]

Requirement: Life cycle management

Definition: Creation and deletion of instances of services taking part in the workflow.

Relevance: Grid. Not yet for web services, but is an issue in WS-Resource Framework

Implications:

Literature: [13, 33]

Requirement: Fault tolerance

Definition: If a service is not executed within a specified time, then fault tolerance will execute. Implicit fault tolerance guarantees a basic fault tolerance and explicit fault tolerance handles user-defined fault management strategies.

Relevance: Grid.

Implications:

Literature: [34]

Requirement: Reliability of individual services and service compositions

Reliability of individual webservices: Traditional fault tolerance solutions are not effective. Two strategies: to reduce MTTF (Mean Time To Fail) AND to reduce MTTR (Mean Time to Recover)

Reliability within a service composition. No standards (e.g. WSDL, SOAP, nor WSFL or XLANG) does address reliability/failed transaction across multiple constituent services. (Q: *When a service*



composition recovers, where does it start? From the beginning or where it crashed (as close to that point in the execution as possible?).

Definition: .

Relevance: Grid, p2p, web services.

Implications:.

Literature: [35]

Requirement: Workflow granularity

Definition: The ability to specify both an abstract and a concrete workflow. In the abstract workflow there are no bindings to specific service implementations, which is the case for concrete workflows.

Relevance: Grid.

Implications: An abstract workflow uses logical names possibly connected to ontologies, while a concrete workflow refers to specific network locations.

Literature: [36]

Requirement: QoS optimization criteria

Definition: The user should be able to specify the QoS optimization criteria such as minimizing execution time and the use of expensive resources.

Relevance: Grid.

Implications:

Literature: [36]

Requirements: Parallel Loops

Definition: Parallel loops can handle the execution of a large number of independent tasks.

Relevance: Grid.

Implications:

Literature: [32]

Requirement: Discriminator control-flow pattern

Definition: A number of parallel activities are executed and the first one to return an answer is used, the others are ignored.

Relevance: Grid and Web Services

Implications:

Literature: [2, 32]

Requirement: UDDI-enabled peer registries

Definition: Peer descriptors are published through an UDDI-enabled peer instead of publishing each p2p service directly.

Relevance: P2P and Web Services

Implications:

Literature: [37]