

THESIS ON INFORMATICS AND SYSTEM ENGINEERING C65

Composition of Web Services on Large Service Models

RIINA MAIGRE

TUT
PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Institute of Cybernetics

Dissertation was accepted for the defence of the degree of Doctor of Philosophy in Engineering on 2 May 2011.

Supervisors: D.Sc., Leading Researcher Enn Tõugu, Institute of Cybernetics at Tallinn University of Technology

Ph.D., Senior Researcher Peep Kõngas, University of Tartu,
Faculty of Mathematics and Computer Science,
Institute of Computer Science

Opponents: Dr. Margus Veanes, Microsoft Research, Redmond, WA, USA

Prof. Merik Meriste, Faculty of Science and Technology,
Institute of Technology, University of Tartu, Tartu, Estonia

Defence of the thesis: 22 June 2011

Declaration:

Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

/Riina Maigre/



Copyright: Riina Maigre, 2011

ISSN 1406-4731

ISBN 978-9949-23-122-5 (publication)

ISBN 978-9949-23-123-2 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C65

Veebiteenuste kompositsioon suurtel teenustemudelitel

RIINA MAIGRE

Contents

Introduction	7
Problem Statement and Contributions	8
Automated Web Service Composition	9
X-Road	11
References to the Published Work	12
Organisation of the Thesis	13
1 Related Work	15
1.1 Automated Web Service Composition Approaches	15
1.1.1 Automated Web Service Mapping into Service Template	15
1.1.2 Automated Workflow Generation from Existing Web Ser-	16
vices	
1.1.3 Tools that Combine Mapping and Automated Composition	18
1.2 Non-automatic Web Service Composition Tools	19
1.3 Composition on X-Road	21
1.4 Summary	21
2 Service Model and Workflows	22
2.1 Service Model	22
2.2 Hierarchical Service Model	24
2.3 Automated Service Model Generation	25
2.3.1 Using Service Descriptions for Service Model Generation	25
2.3.2 Problems in Web Service Descriptions	27
2.4 Higher-Order Workflow	28
2.5 Service Models and Higher-Order Workflows	29
2.6 Conclusion	29
3 Logic for Composition on Service Models	31
3.1 Preliminaries	32
3.1.1 Implicative Fragment of the Intuitionistic Propositional	32
Logic	
3.1.2 Structural Synthesis of Programs (SSP)	33
3.2 Services and Simple Workflows	35
3.3 Logic for Higher-Order Workflows	36
3.3.1 Control Flow	36
3.3.2 Control and Data Flow	36
3.4 Logical Representation of Service Models	38
3.5 Synthesis of Compound Services	40
3.5.1 Defining a Goal	41

3.5.2	Finding a Proof	41
3.6	Conclusion	42
4	Web Service Composition Tool	43
4.1	Architecture of the Service Composition Tool	43
4.1.1	Knowledge System	43
4.1.2	Knowledge Architecture	44
4.1.3	User Knowledge Level	45
4.1.4	Logical Level	46
4.1.5	Service Implementation Level	46
4.2	CoCoViLa	47
4.3	Implementation of the Web Service Composition Tool	50
4.3.1	Composition Packages	51
4.3.2	Specification Language for Web Service Composition	51
4.3.3	Visual Language for Web Service Composition	53
4.3.4	Output Generation	53
4.4	Conclusion	56
5	Web Service Composition on Large Service Models	57
5.1	X-Road Model	57
5.2	Service Model in CoCoViLa	59
5.3	Automatic Handling of a Service Model	61
5.4	Conclusion	66
6	Web Service Composition on Hierarchical Service Models	67
6.1	Hierarchical X-Road Service Model	67
6.1.1	Generation of a Service Model in CoCoViLa	68
6.1.2	Submodels	71
6.1.3	Hierarchical Model	72
6.2	Composition on Hierarchical Service Models	73
6.2.1	Composition without Higher-Order Nodes	74
6.2.2	Composition with Higher-Order Nodes	77
6.3	Conclusion	80
	Conclusions and Future Work	81
	References	83
	List of Publications	90
	Annotatsioon	91
	Abstract	92
	Elulookirjeldus	93
	Curriculum Vitae	96

Introduction

Web services – software components, accessible over the web using machine processable interface descriptions – have made their way into our everyday lives. Web services are an integral part of Estonian e-government [20, 21]. Initiatives exist for pan-European electronic cooperation, for example, ISA project, which stands for Interoperability Solutions for European Public Administrations [71] and SemanticGov [60]. Web services are offered by major companies, such as Amazon [1] and integrated into various tools for software development and for science and research. For example, web service development and automated service description generation for web services is built into Microsoft .NET software development platform. Web service development is also supported by Java by JAX-WS API.

Web services offer for businesses a possibility to make the developed functionality available to others over the web. For researchers, web services offer an easy way to share data and knowledge with their colleges in other places of the world. They offer a way to decrease doing the same work again and again and to strive for quicker results for both – business and research. Web services can also be used as one of the building blocks of Cloud computing technology (e.g., to offer Software as a Service) which is a growing trend on how services should be created, offered and used in the Internet [15, 52].

An atomic web service usually offers very basic functionality. Quite often the functionality allows, for example, to retrieve or to update some information in the database. Having a number of atomic web services with matching inputs and outputs, bigger and more complex web services can be built from atomic web services. Even simple business transactions and scientific tasks usually include more than one query or update. With web services these tasks can involve different databases and information systems, offered by different organisations, and located on different servers. Seen in this light, it is very important to be able to compose atomic web services into new, more complex web services. Web service composition is a task of constructing new compound web services from a predefined set of atomic or compound existing services. In the process of composition, control constructs, such as cycle and condition, can be added to create the workflow. Compound services are also called complex services in the literature. Standards for web service composition, such as WS-BPEL [64], WS-CDL [65], WSML [61] and OWL-S [42] have been proposed and work well when workflows and the number of web services to choose from are small and when the domain is simple enough for the developer, who is an expert in software development and standards, to understand.

In order to compose web services offered by different organisations, it might be necessary to understand very specialised concepts and functionality of services from a number of different information systems. These information systems might use different ontologies. In a very specialised and knowledge intensive domain, a domain expert's knowledge might be needed in order to construct functional and correctly working workflows. Domain expert, however, may not be a programmer or expert in all web services related standards. In this case, having a simple domain-specific language to specify tasks and problems in the domain would speed up the prototyping and development process. On the other hand, if the number of available web services is very large, that is, thousand or more web services, finding only one missing piece into a manually developed workflow can be complicated even for an expert as there may be many candidate web services with similar functionality. Constructing a complete workflow from scratch is even harder as the complexity of finding suitable web services into a workflow grows with every additional missing piece.

Web service composition is a complicated task, because the domain where the composition is applied may be knowledge intensive, the number of candidate services can be very large and services available for composition can be updated or removed [51]. Therefore, combining automated workflow creation that simplifies finding candidate services with a simple visual language that enables domain experts to manipulate workflows would greatly reduce the complexity of web service composition process. The problem of automating the web service composition process in a knowledge intensive domain with a large number of candidate services by combining planning with visual representation of services is addressed in this thesis.

Many methods have been proposed that either try to automate compound web service creation [45, 51] or offer manual visual modelling of web services [11, 72]. However, there are only a few tools that combine visual language with planning options, for example, Synthly [2], JOpera [44] and Web Service Composer [55]. These tools allow to model only the workflow of the currently developed compound web service. The same thing applies for other visual web service modelling tools – they are used to work with one service at a time, while in the approach proposed in this thesis, all available web services and control constructs are modelled as a service model. A desired compound service is extracted out of this service model automatically using logic-based methods. This offers visual specification possibilities for non-programmer users and allows to translate service models into logic for automated planning.

Problem Statement and Contributions

The problem of automating web service composition in a knowledge intensive domain with large number of available web services that might use different

ontologies is investigated in this thesis. The goal of this work is to propose visual specification possibilities for domain experts, to be able use logic-based methods in order to compose new web services automatically and to apply the composition method to real world knowledge intensive domain with large number of services offered by different organisations. This goal is achieved by creating graphical service models from existing services. Graphical service models are translated into logic that enables to reason automatically about the existing web services. This method is applied to Estonian e-government services.

The following are the results of this thesis:

- The service model concept is proposed and discussed as a way to describe a large number of available web services, control constructs and concepts from an ontology. Service models have a visual representation that enables domain experts, who are not programmers to compose new web services.
- Applicability of intuitionistic propositional logic for describing service models that include control constructs and for synthesis of new compound web services by means of structural program synthesis is investigated.
- To test the handling of large realistic service models, experiments were done on real world, large and knowledge intensive service domain – Estonian e-government web services. Experiments with the web service composition tool implementing the approach described in the thesis are conducted on a large service model based on Estonian e-government web services. These experiments show that the proposed method is applicable to very large service models.
- In order to reduce the complexity on large service models, experiments were performed on hierarchical service models, where web services are separated into different submodels by their providers. A provider is expected to use the same ontology for all web services. However, different submodels can use different ontologies. Models are generated from the real world web service descriptions of the Estonian e-government information system. This work also identifies some problems that one might encounter when trying to automatically generate service models from existing descriptions.

Problems of security and confidentiality that are big concerns when considering web service composition on Estonian e-government web services are out of the scope of this thesis. In addition, this work does not deal with pre- and postconditions in terms of resources, because the example domain – Estonian e-government information system – is mostly data-oriented.

Automated Web Service Composition

The goal of the web service composition is to construct new web services from existing ones. These new web services offer some new functionality that none

of the existing web services provide, but is achievable by combining multiple existing web services and adding some control constructs. The goal of automated web service composition is to automate the process of creating new web services as much as possible. Automated web service composition assumes three steps:

1. description and annotation of web services;
2. finding of web services;
3. building a compound web service.

Web service description and annotation are strongly related to finding of web services. In order to find suitable web services, good descriptions must exist. To solve the problem of web service finding, web services' descriptions must be enhanced with machine-processable semantic web service descriptions. Languages, such as OWL-S [42], SAWSDL [53], WSMML [61] and WSDL-S [63] have been proposed for this. In theory, there should be only a small step from semantic web services to fully automated web service composition. In practice, there are not many web services with semantic descriptions available and there is no easy way to annotate all existing web services such that their semantics would be the same for everyone. This is a big obstacle for the automation of web service composition. For this reason, it has been suggested that it will only be possible to create a common understanding about web services published through one service trader or a service park [46]. It is assumed in this work that web services offered by one provider have a common ontology.

Once a common understanding exists, it is possible to start automating the process of building compound web services. To build compound web services automatically, the user defines a goal that describes the service to be composed without defining the actual component web services. If it is possible to build such a service from existing web services, its structure, that is, a workflow describing the structure in which component web services should be executed, will be automatically generated by the web service composition tool.

Tools for automating web service composition can be divided into two categories. In the first category, there are tools that, given a goal and descriptions of available web services, try to synthesise the complete workflow of a compound web service from scratch. In the second category, there are tools that, given a goal and a workflow containing abstract web services, try to find and map concrete web services into the given workflow. These tools are meant to work with web services annotated with concepts from one ontology and not to address the problem of composing web services offered by different providers or through different service parks that are using different ontologies. An attempt has been made in this work to solve the problem of connecting different web services offered by different providers or different service parks by using hierarchical service models.

Web service composition method proposed in this work is based on the well researched method – structural synthesis of programs (SSP) [38, 40]. The main

idea of structural synthesis of programs is to construct programs automatically from existing preprogrammed components based on their structural properties and a goal given by the user. To construct a new program, its specification, including the goal, needs to be given. If the problem described is solvable using the specification, a constructive proof that a solution to the problem exists is found. After that, a program solving the problem is extracted from the proof. Therefore, the steps from the specification to the program are:

specification and a goal \rightarrow proof \rightarrow program

These steps are exactly the way automatic web service composition could be handled. Preprogrammed components are atomic web services. Defining their input and output variables describes their structural properties. The goal in the case of web service composition is to get a description of the compound web service. If the compound web service requested by a goal can be constructed using the existing web services, a proof of its existence is obtained. A program is extracted from the proof. This program can then execute or simulate the execution of the resulting composition or generate the description of the compound web service in some language designed to describe compound web services, such as BPEL.

X-Road

Web service descriptions of the Estonian e-government information system were used in the experiments described in this thesis. The Estonian e-government information system integrates over 2000 web services from more than 80 different information systems [49]. This means that it is a large and knowledge intensive domain, where both domain expert knowledge and automation of composition would offer new possibilities and reduce work.

The Estonian e-government information system has a service oriented architecture and its central part is the X-Road data-exchange layer [20, 21, 73]. Estonian e-government web services enable to read and write data to and from the national databases and develop business logic based on data. In X-Road, the data exchange is handled by SOAP messages, web services are described in WSDL and web service descriptions are published in a UDDI repository. The descriptions of Estonian e-government web services are not created with web service composition in mind. This makes them an ideal test-bed for identifying problems that might occur in web service composition.

X-Road guarantees secure access to nearly all Estonian national databases over the Internet. It is the environment through which services are provided to the citizens, entrepreneurs and public servants on the 24/7 basis. These services are available through governmental portal www.eesti.ee to a variety of user groups (citizens, entrepreneurs, public servants). The number of requests per month exceeds currently 3 million.

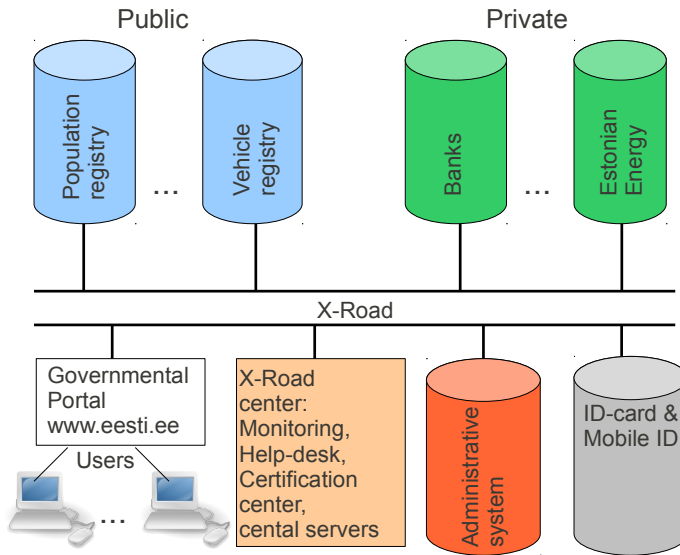


Figure 1. X-Road connects public and private information service providers

Figure 1 shows a simplified structure of the Estonian e-government information system. As demonstrated in the figure, X-Road connects besides public databases also some private ones. These are, for instance, main banks and some privately owned infrastructure enterprises. X-Road infrastructure also offers PKI infrastructure, help-desk and monitoring. Users connect to the system through governmental portal where they can execute predefined services. All queries have to be done one by one, even if semantic connections exist between the underlying web services. The original and more detailed scheme [12] can be found from the homepage of Estonian Informatics Center. For brevity, the Estonian e-government information system web services are from now on called X-Road.

Different information systems offer web services with different functionality and purpose. Currently no joint ontology exists for all X-Road web services and even if it would exist, it is hard to imagine users being able to orient in all concepts of all information systems which differ from one another quite a lot.

References to the Published Work

This section gives an overview of how the chapters in the thesis relate to the author's published work.

The overview of automatic composition tools in Chapter 1 is published as a conference paper “Survey of the Tools for Automating Service Composition” [30] in *The 8th IEEE International Conference on Web Services (ICWS 2010)*.

Chapter 2 that gives a definition for the service model, its hierarchical representation and relation to higher-order workflows is covered in the conference paper “Composition of Services on Hierarchical Service Models” [34] that will be published in *The 21st European - Japanese Conference on Information Modelling and Knowledge Bases (EJC 2011)*.

Chapter 3 on logic for service models is partially based on the conference paper “Compositional Logical Semantics for Business Process Languages” [37] presented at *The 2nd International Conference on Internet and Web applications and Services (ICIW 2007)* and on the journal article “Dynamic Service Synthesis on a Large Service Models of a Federated Governmental Information System” [33] published in *International Journal on Advances in Intelligent Systems*.

Chapter 4 presents the architecture of the composition tool and is based on the conference paper “Stratified Composition of Web Services” [31] presented at *The 8th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2008)*.

Chapter 5 about the experiments done on large service models is based on the conference paper “Handling Large Web Services Models in a Federated Governmental Information System” [32] presented in *The 3rd International Conference on Internet and Web Applications and Services (ICIW 2008)* and on a journal article “Dynamic Service Synthesis on a Large Service Models of a Federated Governmental Information System” [33] published in *International Journal on Advances in Intelligent Systems*.

The experiments on hierarchical service models described in Chapter 6 will appear as a conference paper “Composition of Services on Hierarchical Service Models” [34] in *The 21st European - Japanese Conference on Information Modelling and Knowledge Bases (EJC 2011)*.

Organisation of the Thesis

The thesis is organised as follows. First, an overview of the related work is given in Chapter 1. Second, definitions of the service model and higher-order workflows are given in Chapter 2. Third, logic for service models is described in Chapter 3. A proof of concept tool for composing web services on X-Road service models is described in Chapter 4. Experiments of web service composition on large service models describing X-Road are discussed in Chapter 5 and experiments on hierarchical service models generated from X-Road service descriptions are discussed in Chapter 6.

Acknowledgements

I thank my supervisor Enn Tõugu for encouraging me to do the experiments and in publishing the results. I learned a lot during these four years. I thank my other supervisor Peep Kõngas for introducing X-Road web services to me and for providing the initial model and annotated descriptions of X-Road web services. I am grateful to Pavel Grigorenko, Andres Ojamaa and Professor Merik Meriste for suggestions on improving the thesis.

I thank the Estonian Information Technology Foundation's Tiger University+ programme, Estonian Doctoral School in ICT (2005–2008 and 2009–2015), Estonian Science Foundation grant No. 6886 and the target-financed theme No. 0322709s06 of the Estonian Ministry of Education and Research for financial support.

1 Related Work

This chapter gives an overview of the methods and tools available for automating web service composition. The main emphasis is on methods and tools that automate the process of building compound web services, are suitable for search from large number of web services and are usable by domain experts, who might not be programmers. Unfortunately, little has been published about the scalability of composition methods, and no large examples are given in the literature.

1.1 Automated Web Service Composition Approaches

There are two main approaches to the web service composition: automated web service mapping into service template that was constructed manually and automated workflow generation from existing web services. There are also some tools that combine automated workflow generation with manual workflow construction. All these automation approaches are described in this section, although the combined approach is closest to the composition method described in this thesis.

1.1.1 Automated Web Service Mapping into Service Template

In this approach, a compound service description template is created that describes the workflow and the templates of component web services. Graphical modelling tools are often used for this task. Actual web service implementations are mapped to web service templates when the workflow is executed. Automatic composition in this case means that the mapping of actual web services, that is, finding the web services that satisfy the workflow, is done automatically. This subsection describes tools in this category.

eFlow is a dynamic web service composition and execution system [6] that includes visual compound web service modelling and a possibility to specify web service selection rule in a service node of the model. The suitable web service is found using a service broker.

METEOR-S framework includes a tool for constraint driven web service composition [3]. First, abstract workflow, represented as BPEL process, is manually created in METEOR-S abstract workflow designer. Web services can then be mapped to the workflow by the user or automatically by the system. For mapping web services automatically, service templates need to be defined. A service template includes its data semantics, classification of service operations by functionality, pre- and postconditions and quality of service specifications. Discovery

engine will return a set of service advertisements that match the template. In addition, transformations required to make a service advertisement match the template are queried. At first, all matching web services are selected, then quality of service constraints specified in the abstract workflow are used to find the optimal solution.

Zeng et al. [74] present AgFlow, a middleware platform that enables quality-driven web service composition and execution. State charts are used to represent control and data flow. Web services are assumed to be described in common, community accepted ontology. To select a web service from similar web services offered by one community, quality of service properties are used.

WSMO Studio [8] is a modelling environment for the semantic web services domain. It supports creating and validating WSMO (Web Service Modelling Ontology) [62] models, exporting and importing from various WSML (Web Service Modelling Language) [61] formats, RDF and from a subset of OWL-DL. WSMO Studio includes WSMO Editor that provides a user interface for modelling of WSMO ontologies, goals, and mediators. WSMO Choreography Editor is used to create choreographies that can be executed in the execution environments such as WSMX [18] or IRS-II [9]. Semantic annotations to existing WSDL descriptions can be added using the SAWSDL editor. Repository front-end is utilised for storing and querying WSMO descriptions. WSMO services can be discovered by representing a request in the form of a WSMO goal. A list of matching services is returned as a result.

In all these approaches the workflow has to be created entirely manually even if data dependencies between web services exist. In addition, a web service matching the specified goal might not exist. In this case, the workflow has to be manually redesigned.

1.1.2 Automated Workflow Generation from Existing Web Services

In the second approach, some predefined pool of web services exists (e.g., in a service registry or in a service park), and a goal that has to be satisfied by the compound web service is defined by the user. A solution that satisfies the goal is constructed automatically by a planning method that uses actual web service descriptions as planning data.

Sword [48] is a set of tools for composition and execution of web services. A web service is represented by a rule expressing that given certain inputs, the web service is capable of producing particular outputs. A rule-based expert system is then used to automatically determine whether a desired compound web service can be constructed using existing web services. If it is possible to construct such a web service, a plan for creating a web service is constructed. Execution of the plan instantiates the compound web service. Individual web services are defined by their inputs and outputs. For each web service, a rule is defined.

Rao, Küngas and Matskin [50] describe a system based on propositional linear logic that allows to describe both functional and non-functional properties of web services. DAML-S is used to represent semantic web services. The pool of DAML-S specifications are translated into linear logic axioms. A linear logic theorem prover is used to prove whether the user's defined goal can be achieved by the composition of available atomic web services. If so, the process model is extracted from the proof. The process model of the compound web service can be translated into DAML-S or BPEL. A graphical user interface is used to visualise web services.

Haav et al. [17] propose a framework for informational web services of X-Road. This framework has two parts – annotation part and a logic-based composition part. Annotated web services and a user request are translated into the tool's inner logical language that is then used in the theorem prover RqlGandalf. Result of the synthesis is a Python program.

Kona et al. [23] propose a method that takes a repository of semantically annotated web services and a query to find a directed acyclic graph that represents a compound web service. A resulting composition graph can be translated into OWL-S. A prototype implementation has been tested in the domain of bioinformatics.

ASTRO toolset [47, 56] consists of the following tools: WS-gen – for generating automated compositions; WS-mon – for monitoring; WS-console – extension to the ActiveBPEL execution engine administration console to present the status of monitoring; and WS-animator – to simulate execution. WS-gen takes abstract BPEL specifications of participating partners that are translated into state transition systems for formal description, a choreography specification file and a goal specification file as input. It then uses planning via symbolic model checking technique to generate a state transition system of the process that satisfies the requirements and translates it into BPEL.

QSynth [19] is a tool for quality of service aware automatic web service composition. WSDL and OWL-S files are used to gather web service information which is used to build a dependency graph. The dependency graph is then used to compose new web services. The goal compound web service is defined by its inputs, outputs and quality of service constraints. The resulting compound web service can contain sequence, join and split constructs but cannot contain cycle or condition constructs. The result can be transformed into BPEL. This approach is shown to be scalable to a large number of web services, but seems to work only on atomic web services.

Approaches described in this subsection try to fully automate web service composition. It is easy to apply automated workflow construction when the resulting workflow is a sequence. Automatic synthesis of control constructs or composition of web services that are themselves compound web services, however, needs a very detailed and complex goal specification language and a very detailed goal from the user. Some of these approaches also address this, for

example, ASTRO toolset works with compound web services that are composed to satisfy very detailed goals. However, to specify such goals, a domain expert is needed and specification of the goal might sometimes be more complicated and time consuming than manual specification of a workflow.

1.1.3 Tools that Combine Mapping and Automated Composition

When taking into account that in a specialised and knowledge intensive domain, a domain expert can sometimes design a better workflow than software developers or automated planners, a need for tools that combine both approaches arises. This subsection describes a few tools that try to combine the manual workflow specification with automated synthesis of workflows.

JOpera [44] is a software composition research platform used for visual web service composition, execution and monitoring. JOpera's Visual Composition Language (JVCL) can be mapped to BPEL and vice versa. Building blocks for workflows do not have to be web services. Other invocation mechanisms, such as UNIX shell command execution, RPC and RMI, are also supported. Activities can be imported from UDDI registry by translating WSDL descriptions into JVCL notation. Each web service operation is imported as a separate activity. It is possible to automate data flow creation as the editor can automatically bind parameters with matching names and make recommendations based on the parameter types.

Web Service Composer [55] is a web service composition and execution prototype tool for OWL-S that has two basic components: a composer and an inference engine. Information about known web services is stored in the tool's knowledge base that is used by the inference engine to find matching web services. The inference engine is an OWL reasoner written in Prolog and it is used to find available choices at each composition step. The user starts a composition of a new compound web service by selecting one of the services registered by the engine to be the last web service in the workflow. The rest of the workflow is created backwards by selecting input web services amongst these presented by the tool. This means that new web services providing appropriate input data for the selected web service are suggested automatically. Matching is done using the information given in the service profile. To limit the suggestions found by matching, filtering by non-functional attributes (such as location and type) is supported.

Synthy [2] is a prototype of the semantic web service composition and execution system that aims to combine industry and research approaches. In Synthy the developer has to formally model requirements for the compound web service. Relevant web services to fulfil the tasks are then discovered automatically by the system amongst available semantically annotated web services. If there are no exact matches, web services that could fulfil the task are discovered and a

control flow is created automatically between those web services. The resulting workflow is described in BPEL.

These approaches are most similar to the approach described in this thesis. These tools, however, also have some shortcomings. For instance, workflow generation in JOpera is relying on matching inputs and outputs with the same name, not with the same meaning. In Web Service Composer, matching input web services are discovered automatically, but if there is more than one matching web service, the user still has to make a choice manually. Synthly is the closest to the approach described in this thesis, but it allows to develop one workflow at a time and there is no easy way to change the workflow without designing a new one manually.

1.2 Non-automatic Web Service Composition Tools

This section will cover some tools that are not exactly automation tools but offer some helping utilities or features for manual composition, like a visual workflow modelling language or a built-in UDDI browser. Some of these features were also present in the tools described in previous section.

ZenFlow [36] is a visual web service composition and execution tool that uses BPEL as an underlying composition language. In addition to creating BPEL workflows, importing workflows composed with other tools is supported. Developer has to know which web services are involved in a workflow and insert them to the workflow. Finding web services is supported by the built-in UDDI browser.

Another application for working with BPEL is Sedna [67] – a web service composition environment for scientists. Execution is supported through the ActiveBPEL engine. In Sedna, BPEL is extended with scientific and domain process execution languages. As scientific workflows can contain thousands of atomic web services, some constructs have been added to Sedna’s visual language for supporting scalability and to decrease web service developer’s work. Additional constructs allow, for example, parallel execution, using a set of activities as a single atomic unit, and the hierarchical composition of workflows. Hierarchical composition allows to describe subworkflows first, and then use them by other workflows. Subworkflows can be described by a WSDL description. This enables other workflows to invoke a subworkflow like any other web service. Hierarchical composition is useful as it reduces the complexity of the workflow for the end user.

Taverna [43] is a tool that allows biologists and bioinformaticians to manually construct workflows from web services and execute these workflows. Workflows created in Taverna are described in Taverna’s internal language ScufI and can be semantically annotated. Taverna provides utilities to locate available web services from provided data sources. It also has the support for workflow execution

monitoring and data provenance that are very important in case of scientific workflows.

Kraemer, Samset and Braek [24] describe a web service orchestration method implemented in the Eclipse-based tool Arctis. Method allows to import WSDL files that become available to the developer as building blocks in UML. This allows developers to work on UML level rather than on web service level. Developed UML compositions are validated by model checking and transformed into executable state machines. The system supports automatic code generation for different frameworks and platforms.

Triana [35] supports the graphical composition and distributed execution of services. Triana communicates with services through the interface that has bindings for multiple middlewares (e.g., JXTA, Web services, OGSA). The exact functionality of advertising, locating and communication with other services depends on the binding. In case of web services, for example, discovery means keyword-based search from the UDDI registry. Services are invoked through a gateway that also handles data type conversions. Services retrieved from the UDDI can be used in composition by dragging them from the toolbox to the canvas and connecting them. Looping and conditional constructs are supported to define the control flow. Constructed compound services can be executed or saved in Triana's custom XML or BPEL.

OWL-S Editor [54] supports importing WSDL descriptions and adding necessary semantical annotations through a user interface. Atomic processes and their inputs and outputs are extracted from WSDL. UML activity diagrams are used for visual composition of web services.

Protégé ontology editing framework includes an OWL-S plug-in [13] that has a graphical, form-based user interface for describing web services in OWL-S and executing them in the editing environment. It is also possible to visualise the specified control flow and data flow as a UML activity diagram. However, the visual representation is only for visualisation and cannot be modified by the user. Parts of the OWL-S description can be generated automatically based on the inputs and outputs defined in the WSDL file.

WSMT [22] is a framework for developing ontologies, web services, goals and mediators based on WSMO. It provides different WSML editors and conversion tools to RDF and OWL. Services can be executed in the semantic execution environments such as WSMX. The framework provides text-based and form-based WSML editors. A graph-based visualiser that allows direct editing is also included.

Liu, Huang and Mei [29] describe an approach that allows users to compose web services as mashups in the web-based composition tool. The user starts with a keyword-based search that brings out suitable web services, from which the user has to make a choice. To include next web service into mashup, background analysis information is used to suggest suitable web services. Background analy-

sis is done by building a directed acyclic graph based on input/output values and annotation information extracted from WSDL.

1.3 Composition on X-Road

In addition to all the development done on X-Road [20, 21], X-Road web services have been used, for example, in web services research and in testing web service composition frameworks. A brief summary of these works is given here.

Automated composition of X-Road web services was proposed and tested by Haav et al. [17]. Kúngas and Matskin used automated composition methods on X-Road web services for analysing differences between governmental and commercial domains [26, 27]. The initial large X-Road model, used in the present work as a base model for experiments in Chapter 5, is derived from the work by Kúngas and Matskin [27]. Kúngas and Dumas [25] used X-Road web services to test their automated annotation method.

1.4 Summary

The current state of tools and methods that increase the automation of web service composition was given in this chapter. Although the number of tools created for simplifying and automating web service composition is large, the problem of automated composition is not solved and complete automation has not been reached.

Tools that offer manual workflow specification are hard to use when the number of web services is large. Things are a bit easier, when web services are automatically mapped into manually generated workflow. However, when it appears that there are no web services that match the workflow then the workflow has to be manually redesigned. Methods that enable automated workflow synthesis from existing web services either work only with atomic web services or need very detailed and complex specifications of goals to compose two or more compound web services. Specifying very detailed goals might be harder than developing the workflow manually. Therefore, for some domains, there is a great need for tools that combine manual workflow specification with automated workflow synthesis. Combined tools should offer visual interface for manual workflow construction and a reasonably fast planner that offers automated synthesis in case of large number of web services.

2 Service Model and Workflows

The concept of service model [34] is proposed in this thesis as a way to describe a large number of available web services and concepts from ontology. Service model is a central part of the web service composition method developed in this work and it is explained in this chapter. First, the concept of service model is defined. Second, hierarchical version of the service model is explained. Third, possibilities to automate the service model generation are discussed. Fourth, the concept of higher-order workflow suitable for representing service models with cycles and other control nodes is introduced. Finally, service model's relation to higher-order workflows is described.

2.1 Service Model

Service model is a description of a collection of interoperable services that includes information necessary for automatic composition of compound services and uses one ontology. Service model can describe a service park. Services in one service model have to share a common ontology and background information. Services from different service parks that share a common ontology can be represented in one and the same service model. Service models that use different ontologies can be connected using hierarchical representation of the service model. This is discussed in Subsection 2.2. Note that, although web services are used in the experiments described in Chapters 5 and 6, the concept of service model is also applicable to services that are using other protocols than SOAP or to custom services that are not made available on the web. Experiments with custom services have also been done as part of this work [34].

A service model is abstractly represented as a bipartite graph with two sets of nodes R and V . The set R is a set of services and data dependency relations that can also be represented by atomic services. The set V is a set of variables representing data that can be inputs and outputs of services, and logical variables that are pre- and postconditions of services. Elements of V have names from the ontology used. A node v of V is bound with a service r of R by an arc (v,r) if and only if it is an input or precondition of r , and with an arc (r,v) , if and only if it is an output or postcondition of r . Service model can include higher-order nodes, for example, cycle and condition constructs or services that are already composed from other services. This will require some marking of arcs. The higher-order nodes will be discussed in Section 2.3 and in Chapter 3. Example of the service model is shown in Figure 2.1, where rectangles represent service

nodes and circles represent variables. Filled rectangle represents a higher-order node.

A data dependency relation represents a connection between variables:

- with the same type, if background information (i.e., the information given by a domain expert) shows that they have the same meaning;
- with different types or format, but the same meaning, if they can be connected using a transformation component.

A two-way data dependency relation between any $v1$ and $v2$ can be always represented by two abstract services r' and r'' and arcs $(v1,r')$, $(r',v2)$, $(v2,r'')$, $(r'',v1)$. An abstract service is implemented by the synthesiser, it does not have an explicit grounding. If v is a data structure, containing other variables, for example, v' , v'' and v''' , then a data dependency relation r can be used to extract the elements out of the structure v . Data dependency relation that takes the elements $v8$, $v9$ and $v10$ out of the structure $v7$ is represented as a triangle in Figure 2.1. It can be used also as a constructor of $v7$ from $v8$, $v9$ and $v10$.

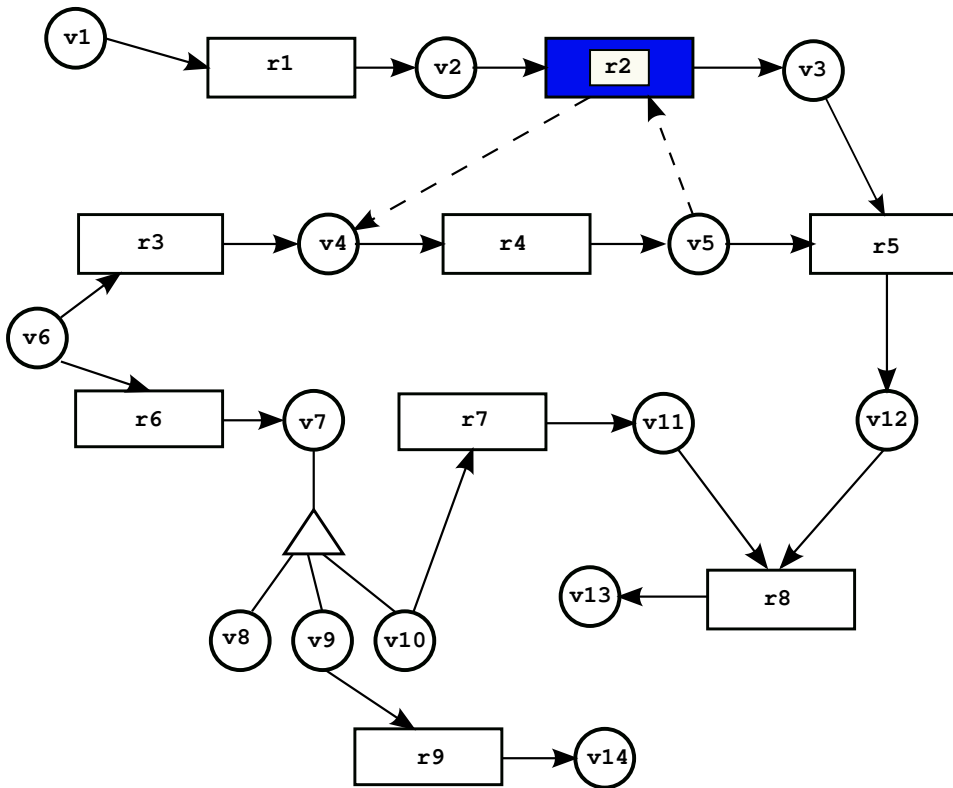


Figure 2.1. Service model represented as a graph

There is a number of ways the service model can be represented (not only as a graph, but also as a set of formulas), the only restriction is that it has to be automatically processable in order to enable automated service composition. Service model's representation as a set of formulas is described in Chapter 3. Compound service will be correct with respect to the service model, if the model is used for composition by a provably correct synthesiser.

The difference between the service model approach and workflow-based composition approaches, for example, BPEL, is that although a workflow can be quite complex, it still represents one possible combination of services. Service model, on the other hand, is a universal description of a service park that describes all currently possible workflows, and the choice of a concrete workflow depends on the goal that a compound service has to serve.

The service model approach differs from most logic-based approaches because, in addition to obvious input and output mappings, domain's background information, given by an expert, can be taken into account when creating a service model. For example, new data dependency relations can be added to connect either data entities with the same meaning but different names or data entities with different types, if type transformation is possible. Besides that, the user is allowed to extend the model with customised higher-order nodes or atomic services. That is, service model's description is not limited to usual workflow constructs like sequences, cycles and conditions, but developers can create services that, for example, count the elements returned as a result of the invocation or take some elements out of the set that is returned as a result. Of course, in order to share the model with other organisations, the functionality of these services must be made available over the web and custom higher-order components need to be translatable into BPEL constructs in order to generate BPEL as an output.

2.2 Hierarchical Service Model

The service model that was used in the first experiments in this work was a large flat model representing a part of the X-Road services. This model had almost 900 components. It contained 300 atomic services and about 600 data entities. Experiments on this model are described in Chapter 5. The visual representation of this service model was very hard to use. Therefore, it was necessary to reduce the complexity, without making sacrifices in the expressiveness of the service model. One way to do that was to hide the information needed by different users into different layers by dividing services into submodels and using them as components in the upper layer service model. By using different submodels as components, different views of hierarchical models can be created.

There are different possibilities for grouping services into submodels. The approach taken in this work assumes that services are divided into different

submodels by their provider. A different approach would be to use some web service/operation clustering algorithm (for example, methods suggested by El-gazzar, Hassan and Martin [14] and Nayak and Lee [41]) to group web services by functionality and use these functional groups as components in the upper layer. This would simplify the composition of functionality-driven web services. X-Road web services domain, however, is more data-driven, that is, it is based on information web services, hence grouping by functionality would not add much value. In addition, grouping by provider allows to hide some data resources of the hierarchical component, representing the submodel, from the user of the upper layer. For example, it is possible to hide data entities that are used only by one provider and therefore would not add any new arcs to the upper level. This will make upper layer representation simpler and easier to use. It also enables to use different ontologies for different submodels and to use a simplified joint ontology with less concepts for the upper level model, where only those concepts that are relevant for the user are defined. Experiments with hierarchical X-Road service model are described in Chapter 6.

2.3 Automated Service Model Generation

Creating service models manually is a complicated task and duplicates the work that is already done when creating web services and their descriptions. Therefore, programs that enable to translate web service descriptions to service models were created to automate service model generation as a part of this work. It appeared that in the current state of X-Road web service descriptions it is not possible to fully automate the service model generation. The idea of automated service model generation and the problems that arose are described in this section.

2.3.1 Using Service Descriptions for Service Model Generation

Given a set of semantically annotated web service descriptions, the process of developing a hierarchical service model begins with generating flat service models from web service descriptions. If web service descriptions and semantic annotations are correct and automatically processable, then this part can be fully automated.

Web service descriptions given in SAWSDL [53] – Semantic Annotations for WSDL, were used to automate service model generation. In this setting, web service operations in SAWSDL are translated into atomic services in the service model, concepts in the ontology are translated into data entities and messages are mapped to arcs. If necessary, data dependency relations are added. Service models generated in this way are used in Chapter 6.

Models created from SAWSDL descriptions can be used as components in the hierarchical service model. Creation of hierarchical service models can also be partially automated, but in general a domain expert is expected to work on this

level. Depending on the context, a domain expert can remove some connections or components and add other connections or components to the generated model. This way it is possible to create views to serve the needs of different users. The relations between semantically annotated web service descriptions, service models and hierarchical views is shown in Figure 2.2.

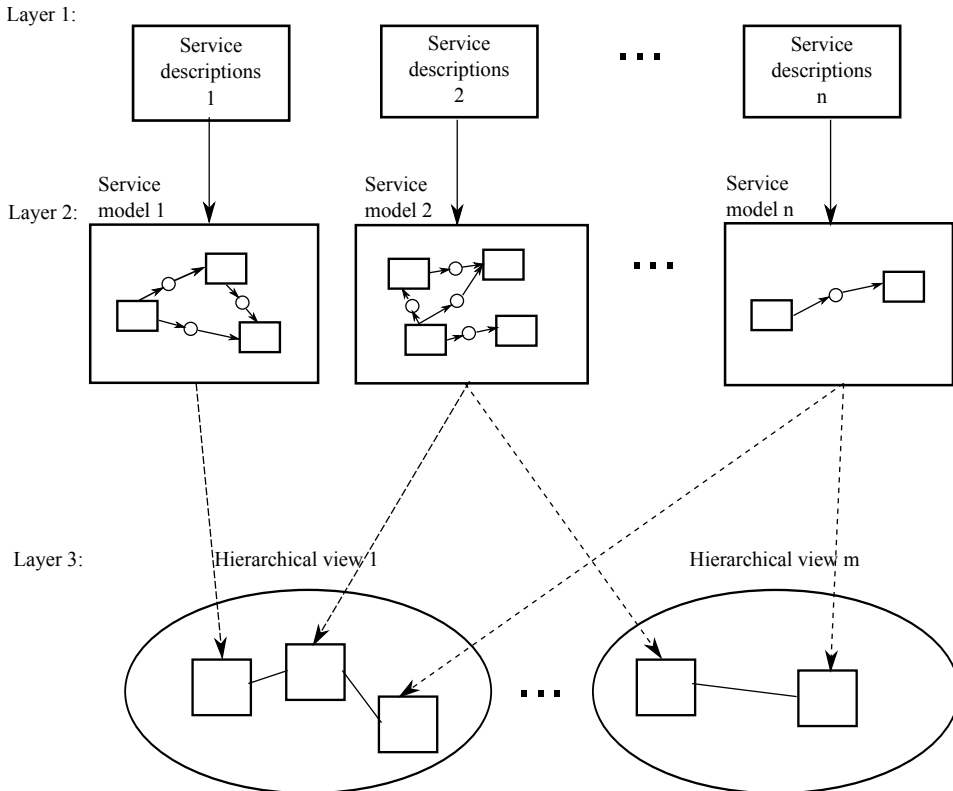


Figure 2.2. Relations between annotated web service descriptions and service models and hierarchical views

In the first layer in Figure 2.2 there are annotated web service descriptions given, for example, in SAWSDL. Service models representing atomic services and data entities are generated from these descriptions. Small rectangles in Layer 2 represent atomic services, circles represent data entities and arrows between them represent data paths. Service models are saved as components and used in a view of a hierarchical model in Layer 3. Note that there can be more than one hierarchical model created from a set of services and that, depending on the task's context, different hierarchical models may contain different service models as components. Therefore, these upper layer service models are called hierarchical views in this thesis.

2.3.2 Problems in Web Service Descriptions

Web service descriptions as well as service models must be correct and easy to use in order to serve their purpose. This section describes some problems that came up when generating service models from existing X-Road web service descriptions. It should be noted that these are just remarks about the experiments and complete analysis of X-Road web service descriptions was not conducted as part of this work.

In order to cover all data dependencies correctly, the common ontology and correct semantic annotations have to exist. X-Road web services are not yet fully annotated and the annotations that exist are mainly syntactic, therefore, after generating a model automatically it has to be revised by the domain expert. The bigger problem is that WSDL descriptions themselves are often automatically generated by the web service development frameworks that create non-explanatory names that rarely contain comments. This kind of web service descriptions are hard to find and use even for human users and it is impossible to find and use these descriptions automatically, unless they contain semantic annotations. In this context, processing these descriptions automatically to generate always correct service models is almost impossible.

Crasso et al. [7] have pointed out common mistakes in WSDL documents. They give six suggestions for revising a WSDL document:

1. separate the schema from the definition of the offered operations;
2. remove repeated WSDL and XSD code;
3. put error information within fault messages and only convey operation results as output of operations;
4. replace WSDL element names with explanatory names if original names are cryptic (this includes the suggestion to write operation names in the form `<verb>+ <noun>`);
5. move noncohesive operations from their original port type to separate port types;
6. document the operations.

These suggestions are meant to improve usability and to simplify finding of web services for human users, but they would improve the possibilities for generating service models as well. Many of those problems are also present in X-Road web service descriptions. For example, X-Road specification of requirements on information systems and adapter servers [68] requires that non-technical faults should be included in the output of the operation and that WSDL fault message should only be used for technical faults. This means that non-technical faults are considered to be results. When generating a service model from the WSDL where error is considered as a result, the resulting model is obviously wrong, as the fault is not a result and it cannot be used in composition.

X-Road WSDL descriptions often use cryptic or ambiguous operation names and it is required by X-Road specification [68] that message parts must be named *paring* (query) or *keha* (body) that says nothing about the actual data and creates redundancy. In addition, only few providers use the form $\langle verb \rangle + \langle noun \rangle$ for operation names. With information web services the number of possible verbs is quite limited. For example, most operations that are done with data could be described with the following verbs *select*, *update*, *delete*, *insert* or their synonyms. These verbs give quite a good idea about the functionality of operations. When a web service provider offers web services through a service park, then it should use the verbs accepted for service description in this particular service park, and therefore limiting the number of verbs used for the same functionality is even easier.

Web service descriptions created by following the suggestions given, for example, by Crasso et al. [7], would make automated generation of service models that represent web services much easier and more accurate. Less cryptic names and documentation would also make it easier for the human domain expert to revise the service models generated from partially annotated web service descriptions. In addition, interoperability between different providers would become easier to achieve as the web services would become easier to find and use.

2.4 Higher-Order Workflow

This section briefly describes the concept of higher-order workflow (HOW). Higher-order workflows enable to represent both control and data flow by a graph. Preconditions will denote from here on both logical propositions and inputs, meaning for an input x that “ x is given”. Respectively for outputs – an output x becomes a proposition “ x is given”. Therefore a service is represented by a node and has connections to its pre- and postconditions. Control flow is represented by means of higher-order nodes that in addition to simple data pre- and postconditions, have services as parameters. Therefore, a higher-order node is represented by a node that has connections to its pre- and postconditions as well as to pre- and postconditions of the parameter services.

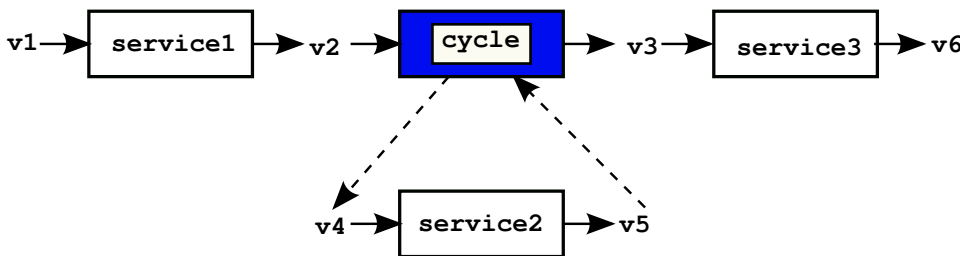


Figure 2.3. Higher-order workflow

Figure 2.3 illustrates a higher-order workflow containing a higher-order node *cycle* (filled rectangle) and three atomic services (white rectangles). This workflow has $v1$ as precondition and $v6$ as postcondition. Service *service2*, with precondition $v4$ and postcondition $v5$, is a parameter for the higher-order node *cycle*. That is, *service2* gets input from the body of a *cycle* and it must return a result before *cycle* can continue. The graphical notation used in Figure 2.3 is described in more detail in Chapter 3, where the logical representation of service models is given.

2.5 Service Models and Higher-Order Workflows

Service models are representable using higher-order workflows. A service model represents all possible combinations of services in the domain. These combinations can contain higher-order nodes and control constructs. Concrete workflow depends on the goal that represents a compound service to be composed. Figure 2.4 shows three possible workflows (marked with bold arrows) of the service model that was shown in Figure 2.1. Other workflows are possible even in this very simple service model. The three workflows shown in Figure 2.4 include the following service nodes:

1. $r1, r2(r4)$;
2. $r6, r9$;
3. $r7, r8$.

The first workflow $r1, r2(r4)$ is a higher-order workflow, with an higher-order node $r2$. This workflow has $v1$ as a precondition and $v3$ as a postcondition. Service $r4$ is a parameter for higher-order node $r2$. The second workflow $r6, r9$ has $v6$ as a precondition and $v14$ as a postcondition. The postcondition $v7$ of the service $r6$ is a data-structure, containing elements $v8, v9$ and $v10$. This structure is connected to the precondition $v9$ of the service $r9$ using the data dependency relation (represented as a triangle). The third workflow $r7, r8$ is a simple workflow without higher-order nodes or data dependency relations. It has $v10$ and $v12$ as preconditions and $v13$ as a postcondition.

2.6 Conclusion

The service model concept was defined and its graphical representation was given in this chapter. Service models can be generated from annotated web service descriptions and represented hierarchically in order to reduce the complexity. This means that the goal to offer visual specification possibilities for non-programmer users is achieved. However, there is still a need to reason automatically about the service models. In order to do this, logical representation of service model is given in Chapter 3.

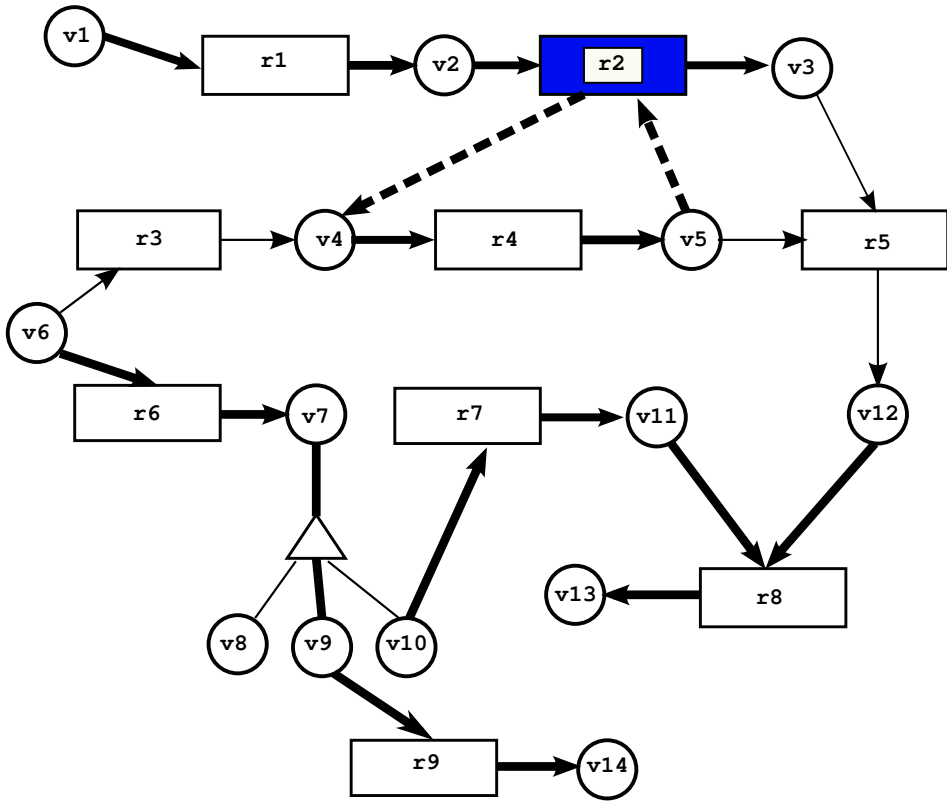


Figure 2.4. Three possible workflows in the service model

3 Logic for Composition on Service Models

The concept of service model was defined and its graphical representation was given in Chapter 2.1. In order to reduce the complexity of the web service composition process, it is necessary to be able to handle service models automatically. Intuitionistic propositional logic [39] is used in this thesis for representing the semantics of service models. Service models are represented by means of higher-order workflows (HOW) and the logic for service models is described by means of logic for higher-order workflows. The logic for representing service models is also applicable to all kinds of services, not just to web services.

Services and control constructs – the nodes in a higher-order workflow can be described by formulas in a particular logic. Proofs in the logic can be transformed into higher-order workflows. This allows to synthesise a new higher-order workflow from a proof. Higher-order workflow can be transformed into some process language (for example, BPEL) via instantiating control and service nodes by the language constructs or by transforming the structure of the resulting higher-order workflow into some process language. This allows one to generate composite processes (compound web services) automatically [33, 37]. Relations between logical representation, higher-order workflow and process languages are shown in Figure 3.1. In this figure HOW represents both a graphical representation of a higher-order workflow and a synthesised structure of a new compound web service, which is itself a higher-order workflow.

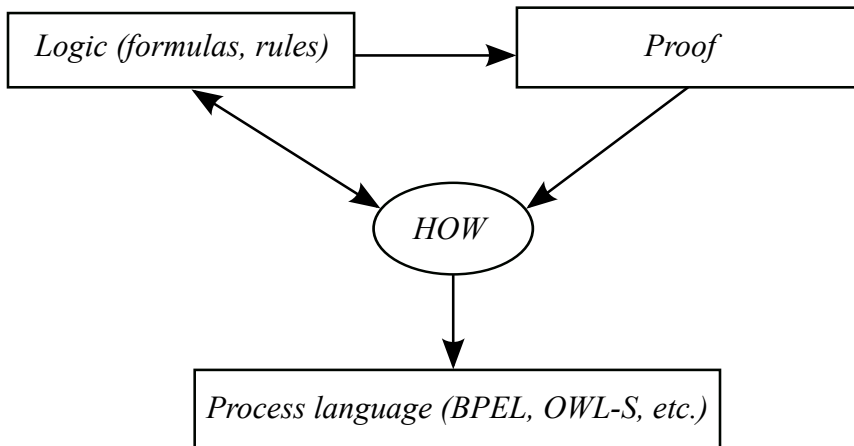


Figure 3.1. Logic, HOW and process languages

This chapter describes logical semantics for representing service models and automating the creation of web service workflows. First, a short overview of the implicative fragment of the intuitionistic propositional logic and structural synthesis of programs is given. Second, a logical representation of services and simple workflows containing only sequences is introduced. Third, logical representation of higher-order workflows for representing control constructs is described. Fourth, the approach is extended to service models. And finally, compound web service synthesis method that uses the intuitionistic propositional logic implemented in structural synthesis of programs is covered.

3.1 Preliminaries

Structural synthesis of programs (SSP) [38, 40, 57] is used for web service composition in this thesis. SSP uses an implicative fragment of the intuitionistic propositional logic. Therefore, it is also the logic used to represent service models in this work. A short reminder of intuitionistic logic and SSP is given in this section. This section is based on works written by Tyugu [59], Mints and Tyugu [40], Matskin and Tyugu [38] and Lämmermann [28].

3.1.1 Implicative Fragment of the Intuitionistic Propositional Logic

In intuitionistic propositional logic (IPL) interpretation of a proposition is not a truth-value, but an object with a proof of its existence. Implicative fragment of the intuitionistic propositional logic uses only introduction and elimination rules for conjunction and implication. Table 3.1 presents the inference rules for the implicative fragment of IPL, where A and B are metavariables for propositional formulas and Γ is a list of formulas.

Table 3.1. Inference rules for the implicative fragment of IPL

Introduction rules:	Elimination rules:
$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge I$	$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge E_L, \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge E_R$
$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B} \supset I$	$\frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B} \supset E$

In intuitionistic logic the information about constructing an object can be extracted from the proof of its existence. To make this explicit, realisations can be added to propositions. In this case, $a : A$ means that a is a realisation of the formula A . Table 3.2 shows the rules of implicative fragment of IPL with realisations.

Table 3.2. Inference rules for implicative fragment of IPL with realisations

Introduction rules:	Elimination rules:
$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \wedge B} \wedge I$	$\frac{\Gamma \vdash f : A \wedge B}{\Gamma \vdash fst f : A} \wedge E_L, \frac{\Gamma \vdash f : A \wedge B}{\Gamma \vdash snd f : B} \wedge E_R$
$\frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash (\lambda a. b) : A \supset B} \supset I$	$\frac{\Gamma \vdash f : A \supset B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \supset E$

In conjunction introduction rule, a represents the realisation of A and b represents the realisation of B . Realisation of the conjunction $A \wedge B$ is represented by the pair (a, b) . In conjunction elimination rules $fst f$ extracts the realisation for A from f and $snd f$ extracts the realisation for B from f . In the implication introduction rule realisation of A becomes a variable bound by λ on the realisation of B . In the implication elimination rule a function f (that is a realisation of $A \supset B$) is applied to a .

3.1.2 Structural Synthesis of Programs (SSP)

The main idea of structural synthesis of programs is to construct programs automatically from existing preprogrammed components based on their structural properties and a goal the user wants to achieve.

Instead of the conventional specification $\forall x(P(x) \supset \exists yR(x, y))$ of the program component where input x and output y are bound by the relation R , SSP uses the specification $\forall x(P(X) \supset \exists yR(y))$. This allows to consider the closed formulas $\exists x(P(x))$ and $\exists yR(y)$ as propositions and the specification becomes $P \supset R$, where P denotes the computability of the input and R denotes the computability of the output. This means that logical language of SSP is an implicative fragment of propositional language with restricted nestedness of implications. SSP's logical language has three kinds of formulas:

1. Propositional variables: A, B, C, \dots
2. One level implications: $A \wedge \dots \wedge B \supset C$
3. Two level implications:
 $(A_1 \wedge \dots \wedge A_n \supset B_1) \wedge \dots \wedge (D_1 \wedge \dots \wedge D_m \supset B_k) \wedge E \wedge \dots \wedge F \supset C$

Computability statement is a formula that represents a function: $a_1, a_2, \dots, a_n \supset b_1, b_2, \dots, b_m\{f\}$, where a_1, a_2, \dots, a_n are inputs and b_1, b_2, \dots, b_m are outputs of the function f . The function f is actually a realisation of the formula just like it was used in Subsection 3.1.1.

Computational meaning of the formulas from SSP's logical language is the following:

1. Propositional variable corresponds to the object variable from the specification. If the problem specification includes a variable a , then there exists

- a propositional variable A in the logical language. Propositional variable A expresses the fact that a value of an object variable a can be computed.
2. Formula $A \wedge \dots \wedge B \supset C$ expresses the computability of object variable c (corresponding to the propositional variable C in the logic) from object variables a, \dots, b that correspond to the propositional variables A, \dots, B . Computability of the object variable c from the object variables a, \dots, b by a function g can also be expressed as a computability statement $a, \dots, b \supset c\{g\}$, where the inputs of the function g is on the left side of the arrow and the output is on the right side of the arrow.
 3. Formula $(A_1 \wedge \dots \wedge A_n \supset B_1) \wedge \dots \wedge (D_1 \wedge \dots \wedge D_m \supset B_k) \wedge E \wedge \dots \wedge F \supset C$ expresses computability of object variable c (corresponding to propositional variable C) from variables e, \dots, f when values of object variables b_1, \dots, b_k are computable from their inputs a_1, \dots, a_n and d_1, \dots, d_m respectively. Computability statements that other computability statements depend on are called subtasks. For example, computability statement $a_1, \dots, a_n \supset b_1$ is a subtask of the computability statement: $(a_1, \dots, a_n \supset b_1), d, e \supset c\{g\}$. Note that realisation of the subtask is not given in the formula, it must be synthesised.

Only one output was shown in the computability statements above, however, one output was shown for simplicity and it is not a restriction of logic. Therefore, computability statements can have multiple outputs, just like it was shown before.

When using the conventional IPL rules shown in Table 3.2, programs extracted from the proof will contain unnecessary steps of composition and decomposition of data structures that correspond to realisations of conjunctions. This, however, leads to inefficient programs. To solve this problem, SSP uses the following three structural synthesis rules (SSR) that are admissible rules in intuitionistic propositional logic. Note that realisations are given in the curly brackets. Bold capital letter (i.e., \mathbf{A} and \mathbf{C}_i) represent conjunctions and lists.

Implication introduction:

$$\frac{\Gamma, \mathbf{A}\{\mathbf{a}\} \vdash B\{b\}}{\Gamma \vdash \mathbf{A} \supset B\{\lambda \mathbf{a}.b\}} \supset I$$

Implication elimination:

$$\frac{\Gamma \vdash \mathbf{A} \supset B\{f\} \quad \Sigma_i \vdash A_i\{a_i\}}{\Gamma, \Sigma_1, \Sigma_2, \dots, \Sigma_n \vdash B\{f(a_1, a_2, \dots, a_n)\}} \supset E$$

where $i = 1, 2, \dots, n$.

Double implication elimination:

$$\frac{\vdash \wedge_i (\mathbf{C}_i \supset D_i\{\phi_i\}) \wedge \mathbf{A} \supset B\{f\} \quad \Gamma_i, \mathbf{C}_i \vdash D_i\{g_i\} \quad \Sigma_j \vdash A_j\{a_j\}}{\Gamma_1, \Gamma_2, \dots, \Gamma_n, \Sigma_1, \Sigma_2, \dots, \Sigma_n \vdash B\{f'(g_1, g_2, \dots, g_n, a_1, a_2, \dots, a_m)\}} \supset EE$$

where $i = 1, 2, \dots, n$ and $j = 1, 2, \dots, m$.

Logic described in this section can be used to represent service models logically and to synthesise new compound services on them. Logical representation of services, data flow, control flow and service models is given in the rest of this chapter. A compound service synthesis rules and an example derivation are given at the end of this chapter.

3.2 Services and Simple Workflows

In intuitionistic propositional logic the functionality of a web service with preconditions A_1, A_2, \dots, A_n and postconditions B_1, B_2, B_m is representable by the following implication:

$$A_1 \wedge \dots \wedge A_n \supset B_1 \wedge \dots \wedge B_m$$

where $A_1, \dots, A_n, B_1, \dots, B_m$ are propositions and having A_1, \dots, A_n one can obtain B_1, \dots, B_m . According to the standard semantics of intuitionistic logic the meaning of a proposition is not a truth-value, it is an object of a proper type. According to the semantics of intuitionistic logic, the implications have a computational meaning, or in other words – their realisations must be functions. In this case their realisations are services that can be shown as the realisations of the computability statement written in curly braces:

$$a_1, \dots, a_n \supset b_1, \dots, b_m \{service\}.$$

When having more than one web service, for example, $a \supset b\{service1\}$ and $b \supset c\{service2\}$, it is easy to represent the order of these two services by a graph. Figure 3.2 shows the graph representation of these two services. Rectangles in the figure represent services and a, b and c represent data entities. Data dependency between services is created with data entities and arrows that represent pre- and postcondition descriptions.

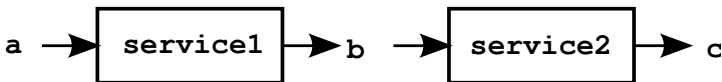


Figure 3.2. Simple workflow with data entities

3.3 Logic for Higher-Order Workflows

The question now is how to represent control constructs, for example, sequence, cycle, condition. The difference between an atomic web service and a control node is that the control node, in addition to simple data pre- and postconditions, has other services and control nodes as parameters.

3.3.1 Control Flow

When two services have to be invoked one after another, they are controlled by a higher-order node *sequence*. This is represented as a graph in Figure 3.3.

The thin line in Figure 3.3 expresses the order of invocation and thick lines express the control that *sequence* has over other services. Types of control nodes with incoming thick lines are one order higher than service nodes at the other end of a thick line, because they have other nodes (both atomic and control) as parameters.

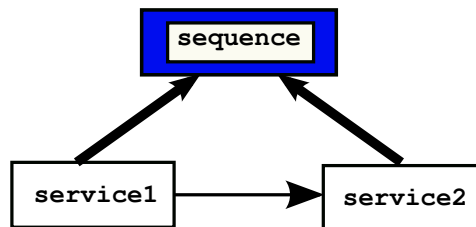


Figure 3.3. Higher-order service *sequence* and atomic services *service1* and *service2*

When a control node has other control nodes as parameters, it can be represented on the graph like shown in Figure 3.4, where control node *cycle* takes *sequence* as a parameter and is itself a parameter to another *sequence*.

3.3.2 Control and Data Flow

To show the data passed between the nodes of the workflow one can extend a workflow graph by including explicit data nodes, and use the arrows as pre- and postcondition descriptions, as it was done with the simple workflow shown in Section 3.2. A higher-order workflow graph extended with data dependencies is shown in Figure 3.5.

The extension by data dependencies adds some implicit control information – order of services may be defined already by data dependencies. When the order is determined by data dependencies, the explicit ordering of control information can be dropped. Deriving control information from data dependencies is the main idea of the composition process described in this work. For example, in the workflow described in Figure 3.5, both *sequence* control nodes, can be dropped

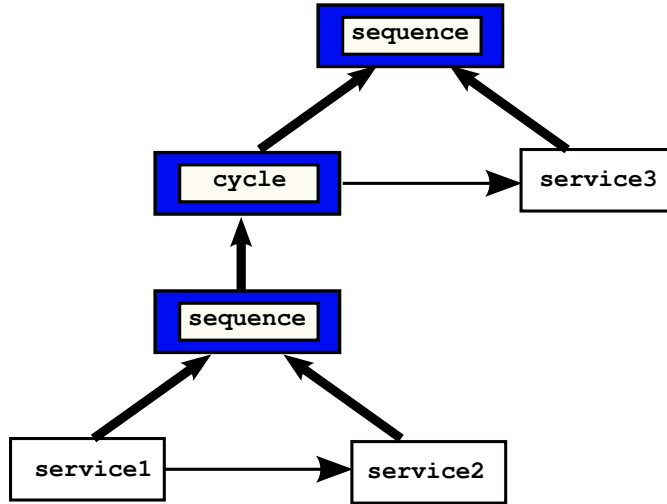


Figure 3.4. Higher-order workflow with three control nodes

because the order is already defined by data dependencies. The result is shown in Figure 3.6.

To avoid operating with names of services in logic, services that are parameters to control nodes can be described by means of their pre- and postconditions, binding them with control nodes, if needed. This is denoted by dashed arrows and logically this is represented by a nested implication. Now the notation must be changed. Instead of thick arrows, thin arrows that have a special marking (dashed arrows) for preconditions and postconditions of nested implications can be used. This way it is possible to use uniform representation for binding data on all workflow models as shown in Figure 3.6. In this figure, b and d express pre- and postcondition of a node (this node is actually a *sequence*, containing two nodes) that is a parameter to the *cycle*.

The logic behind this workflow is expressed by four formulas, one formula for each atomic service and one for a control node. Basic services are described by simple implications:

$$B \supset C\{service1\}; C \supset D\{service2\}; E \supset F\{service3\}.$$

And a control node *cycle* is expressed in logic, using a nested implication as follows:

$$A \wedge (B \supset D) \supset E\{cycle\}.$$

The nested implication $B \supset D$ represents a subtask (i.e., an input) of the control node *cycle*. Realisation of $B \supset D$ is a body of the loop performed by the *cycle*. This body must be synthesised from available services. Higher-order nodes may have more than one subtask and depending on the realisation of the

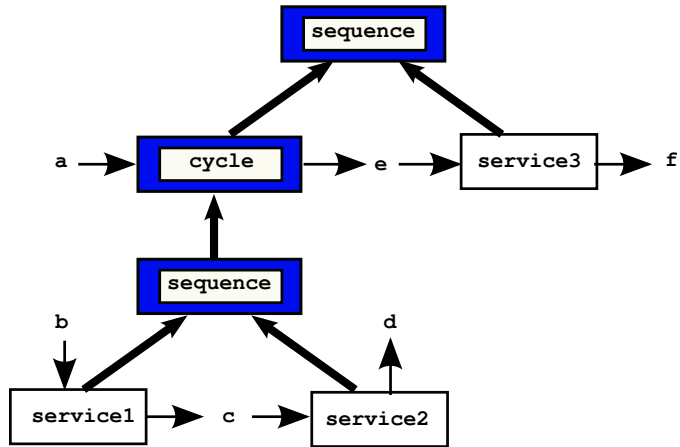


Figure 3.5. Higher-order workflow graph with data dependencies

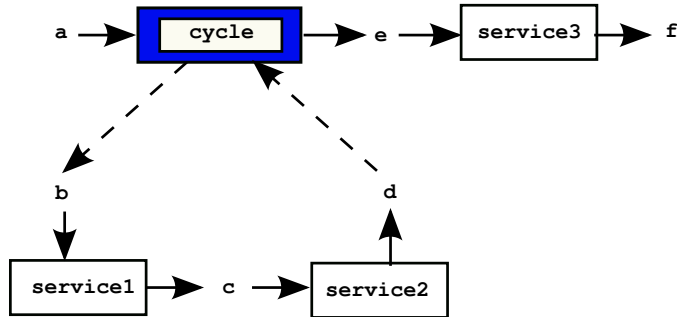


Figure 3.6. Expressing higher-order workflow by means of data dependencies

higher-order node each subtask can be executed multiple times. For example, in case of the higher-order node *cycle*, one might want to perform its body that realises $B \supset D$, until some condition becomes true.

The fragment of logic used here is already sufficient for representing data and control flow and also for synthesising services as nested flows of services taking their data dependencies into account. The evaluation of the condition itself can be expressed by another subtask, if needed.

3.4 Logical Representation of Service Models

It was shown in Section 2.5 that service model is representable by means of higher-order workflows. The same thing applies for the logic. The logic described

in Section 3.3 is applicable to service models. Every service in the service model is described by a formula.

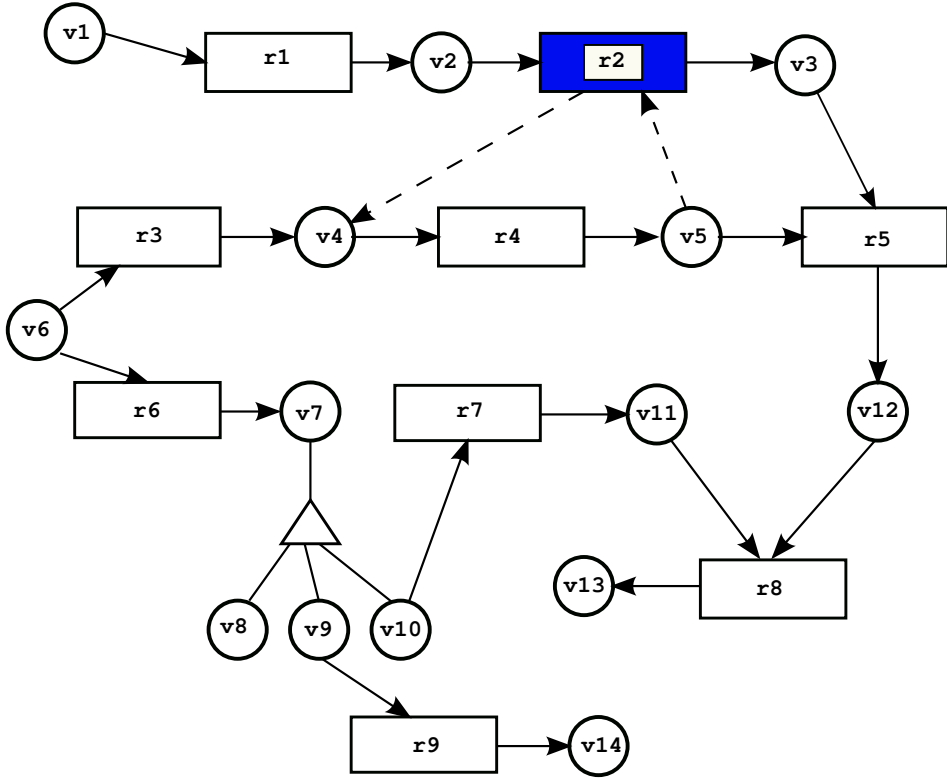


Figure 3.7. Example service model

For example, service model shown in Figure 3.7 (it is the same service model that was shown in Figure 2.1), could be specified with the following implications:

$$V1 \supset V2\{r1\}; V3 \wedge V5 \supset V12\{r5\}; V6 \supset V4\{r3\}; V4 \supset V5\{r4\};$$

$$V6 \supset V7\{r6\}; V10 \supset V11\{r7\}; V9 \supset V14\{r9\}; V11 \wedge V12 \supset V13\{r8\};$$

The implication for the higher-order component *cycle* is the following:

$$(V4 \supset V5) \wedge V2 \supset V3\{r2\};$$

And the implication for the *Selector* component is the following:

$$V7 \supset V8 \wedge V9 \wedge V10\{select\}.$$

The number of atomic services in the service model can be very large. Experiments with service models containing 300 atomic services have been done as

part of this work. These experiments are described in Chapter 5. When hierarchical models are used, components of the upper level model can be unfolded in the higher-order workflow representation. As a consequence, the representation is applicable to hierarchical service models as well. Experiments with hierarchical service models are described in Chapter 6.

3.5 Synthesis of Compound Services

The compound service synthesis method is based on structural synthesis of programs (SSP) [38, 40, 59]. In this setting, a goal service is considered as a computational problem – computing a desired output from a given input. Information for solving the problem is described for SSP by a set of formulas automatically extracted from a specification. A goal that the expected result is computable is formulated as a theorem to be proven. Formulas are given in the intuitionistic logic. A proof of solvability of the problem is built, and a program for solving the problem is extracted from the proof. A tool that implements the method and is based on SSP is described in Chapter 4. The process of automated composition on service models is explained in Figure 3.8.

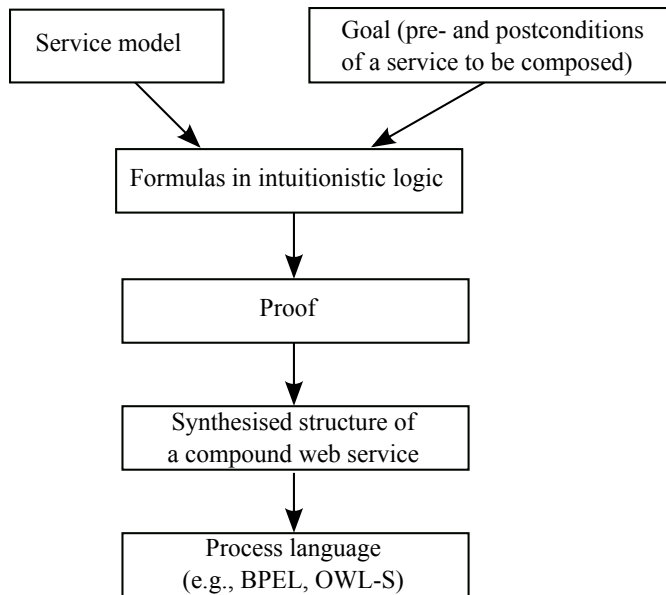


Figure 3.8. Steps in the composition process

In order to compose web services automatically in some domain, using the composition method shown in Figure 3.8, domain’s service model needs to be created. On the service model a goal, describing the pre- and postconditions of a compound web service to be constructed, can be defined by the user. Service

model is then translated into the logic. Using the set of formulas describing the service model, a proof of the composability of the goal service is obtained automatically. The resulting proof corresponds to the structure of the compound service and it can be translated into some process or workflow language.

3.5.1 Defining a Goal

A compound service, that is, a goal, can itself be represented as a logical formula. When a new composite service with preconditions X_1, \dots, X_m and postconditions Y_1, \dots, Y_n has to be built, a goal is given in the form of an implication:

$$X_1 \wedge \dots \wedge X_m \supset Y_1 \wedge \dots \wedge Y_n,$$

A program to construct the goal service is synthesised automatically. On the service model shown in Figure 3.7, a goal can be represented, for example, as an implication $V6 \supset V11$ and on the workflow shown in Figure 3.6 a goal can be represented as an implication $A \supset F$.

3.5.2 Finding a Proof

SSP rules that were given in Subsection 3.1.2 can be simplified for service composition. The SSP rules for service composition are shown here with capital letters as metasymbols denoting conjunctions of propositional variables:

$$\frac{A \supset B \wedge C\{f\} \quad B \wedge D \supset G\{g\}}{A \wedge D \supset C \wedge G\{f; g\}} \quad (\text{SEQ})$$

$$\frac{(A \supset B) \wedge X \supset Z\{f\} \quad A \wedge W \supset B\{g\}}{X \wedge W \supset Z\{f(g)\}} \quad (\text{HOW})$$

The first rule (SEQ) is for sequential composition of services and the second (HOW) is for higher-order control constructs that contain a subtask which can be solved by the program g . A program g can represent, for example, a body of the cycle and must be known before the HOW rule can be applied. When applying HOW, g is given as a parameter for f . The rule HOW contains only one subtask, but formulas with more than one subtask are also admissible.

In case of the example given in Figure 3.6 the problem of computing f from a – the synthesis of compound service with an input a and an output f – is described for SSP with the same formulas that were given before:

$$B \supset C\{service1\};$$

$$C \supset D\{service2\};$$

$$E \supset F\{service3\};$$

$$A \wedge (B \supset D) \supset E\{cycle(\phi)\};$$

and a goal: $A \supset F$.

Note that the realisation of the subtask $B \supset D$ in the program for *cycle*, is unknown and is denoted by the functional variable ϕ . In this example the resulting program is

$$cycle(service1; service2); service3.$$

To derive the goal, the rules SEQ and HOW, must be used. This results in the following derivation:

$$\frac{\frac{A \wedge (B \supset D) \supset E\{cycle(\phi)\} \quad \frac{B \supset C\{service1\} \quad C \supset D\{service2\}}{B \supset D\{service1; service2\}} SEQ}{A \supset E\{cycle(service1; service2)\}} HOW \quad E \supset F\{service3\}}{A \supset F\{cycle(service1; service2); service3\}} SEQ$$

This derivation represents a resulting compound service in logic. It also proves that data entity f is computable from a in the given workflow. It is possible to extract the structure of the resulting compound service from the constructive proof. In addition, realisations given as programs contain the information necessary to either invoke atomic web services or to generate a process language description for a given workflow. The actual result depends on the content of the realisations.

3.6 Conclusion

Semantics of service models, represented by means of implicative fragment of intuitionistic propositional logic was described in this chapter. SSP-based composition on service models forms the web service composition proposed in this work. To investigate the applicability of service models and their logical representation to web service composition, the method was implemented as a proof of concept web service composition tool, which is described in Chapter 4.

4 Web Service Composition Tool

To demonstrate the applicability of the SSP-based web service composition method to service models, a proof of concept tool has been implemented on CoCoViLa model-based software development platform [4, 16]. CoCoViLa supports SSP-based planning and is able to handle specifications in visual or textual form.

Description of the architecture and implementation of the web service composition tool is given in this chapter. First, the knowledge architecture of the web service composition tool [31] is given. Second, model-based software development platform CoCoViLa is introduced. Third, implementation details – composition package, specification language and visual language – of the web service composition tool are described.

4.1 Architecture of the Service Composition Tool

Web service composition is a knowledge intensive process where knowledge is used in different forms and ways. This section gives a brief abstract description of the knowledge usage in the web service composition tool. A concept of knowledge system as a component is used to describe the knowledge architecture of the tool as a composition of knowledge systems.

4.1.1 Knowledge System

The definition of *knowledge system* (KS) and the notations introduced in [58] are used in this thesis to describe the knowledge architecture of the composition tool. A knowledge system is a component – a module of knowledge architecture of a knowledge-based system. It includes a knowledge language, a knowledge handling mechanism, for example, an inference engine, and a method for associating meanings to knowledge objects represented in the knowledge language. It is assumed that there is always a set of meanings and a mapping from knowledge objects into the set of meanings in a knowledge system. The set of knowledge objects S and the set of meanings M of a knowledge system can be represented visually as shown in Figure 4.1.

Knowledge systems can be composed into larger knowledge architectures by connecting them hierarchically, semantically and operationally. Figure 4.2 shows notations for hierarchical and semantic connections of knowledge systems. Two knowledge systems K_1 and K_2 with sets of notations S_1, S_2 and sets of meanings M_1, M_2 respectively are *connected hierarchically* (Figure 4.2(a)), if there is a

relation R between the sets M_1 and S_2 , that is, meanings of the knowledge system K_1 tell something about the knowledge objects of the knowledge system K_2 . K_1 is called upper and K_2 is called lower system. If the relation R is one-to-one mapping between some subsets of M_1 and S_2 , then the knowledge systems are *strongly hierarchically connected*.

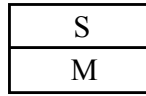


Figure 4.1. Notation of a knowledge system

Knowledge systems K_1 and K_2 are *connected semantically*, if they have one and the same set of meanings M . This is shown in Figure 4.2 (b).

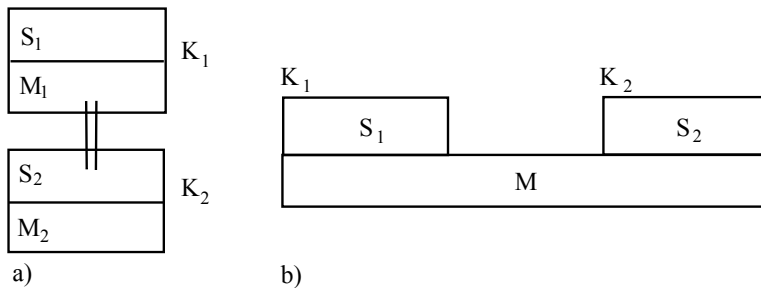


Figure 4.2. Hierarchical (a) and semantic (b) connection of knowledge systems

4.1.2 Knowledge Architecture

The composition tool implemented in this work has three knowledge levels:

1. *user knowledge level* with visual and textual representation of knowledge;
2. *logical level* for formal representation of knowledge and automatic composition of new web services;
3. *service implementation level* for grounding of web services – invoking web services and performing actual computations.

Figure 4.3 shows the knowledge architecture of the web service composition tool developed as a package in CoCoViLa. On the user knowledge level there are two semantically connected knowledge systems: visual and textual knowledge systems. These systems have one and the same set of meanings – unfolded (detailed) specifications of web services. This allows to connect these knowledge systems semantically. Meanings of knowledge objects are unfolded textual representations of specifications on this level. Their elements are almost in one-to-one

correspondence with knowledge objects of the logical level. This makes it easy to introduce the hierarchical connection R_1 between the user knowledge and the logical level. Knowledge objects on the logical level are logical formulas representing functionality of atomic web services, data entities and control nodes. This level is hidden from the user. Meanings of logical level are programs that are realizations of intuitionistic formulas. Implementation of these programs is given in the service implementation level. This creates the hierarchical connection R_2 between the logical level and the service implementation level. Service implementation level also has a hierarchical connection R_3 with the user knowledge level where the grounding information (for example, names and locations of services) comes from. Programs and grounding knowledge objects are mapped into Java programs on the service implementation level. Java program is a generator for the compound service (connection R_4). Meanings of the service implementation level are composed web service descriptions generated by Java programs. The following subsections describe the three knowledge levels in more detail.

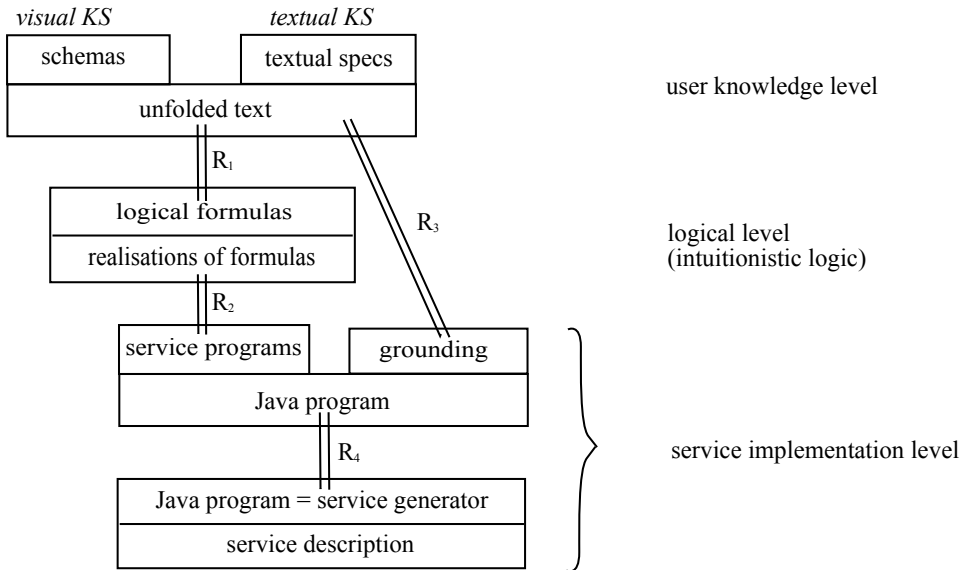


Figure 4.3. Knowledge architecture of the web service composition tool

4.1.3 User Knowledge Level

The user knowledge level supports visual and textual representation of knowledge about service models, handling of inheritance, equality and structural relations. Knowledge objects are both visual and textual specifications of web service composition problems given by users (i.e., service models and goals). This level is implemented in CoCoViLa by extending it with suitable metaclasses for a generic

service. Inheritance, equality and structural relations are supported entirely by CoCoViLa. The user can develop service models manually in the Scheme Editor of CoCoViLa or use service models generated automatically from service descriptions given, for example, in SAWSDL.

It is also possible to introduce values of attributes of components of the model and specify a goal that must be reached by the expected web service. Visual representation of the knowledge on the user knowledge level is shown in Figure 4.4. This figure also shows how a goal is defined using the pop-up window. Textual representation of the knowledge on the user knowledge level is shown in Figure 4.5. The visual model and a textual specification have exactly the same meaning.

4.1.4 Logical Level

Knowledge system of the logical level can be represented as an implicative fragment of the intuitionistic logic, this was explained in Chapter 3. Computability statements describing web services become implications in the logical level and are handled by a theorem prover. The knowledge objects on the logical level are logical formulas – showing the functionality of atomic web services. Meanings of logical formulas on this level are algorithms. A new composite web service is synthesised by means of structural synthesis of programs.

In principle the synthesis process is as follows. The input to the process is a set of logical formulas that represents knowledge about the atomic web services that can be used as components of the synthesised web service, and also a goal, that is, a formula that describes the web service to be composed.

The tool tries to derive logically the goal. If this is possible, then the web service can be composed – the meaning of the derived goal is the algorithm of the expected web service. This algorithm is represented only by its structure. Implementation details are available from the specification given on the user knowledge level, see relation R_3 between the unfolded text and grounding in Figure 4.3.

4.1.5 Service Implementation Level

The main input of the service implementation level is a representation of the structure of a synthesised web service received from the logical level. Another input comes from the specification, as shown in Figure 4.3. These inputs together have meanings as Java programs. The output of this level is a specification of the web service in some process language, for example, in BPEL. This level is a hierarchical connection of two knowledge systems. The higher one takes structure of a web service and grounding of atomic web services, and produces a meaning that is a Java code with a method call for each atomic web service included into the composed web service. The lower one takes the Java code as a knowledge

object and produces a web service description as the respective meaning. That is, first a generator for generating a composed web service is synthesised, and thereafter it is run and a required web service description is generated.

4.2 CoCoViLa

CoCoViLa [4, 16] is a model-based software development environment. CoCoViLa enables the creation, modification and usage of software packages with domain specific visual languages. Each software package allows to create models describing some problem domain. On those models, problems to be solved can be specified. Structural synthesis of programs can then be applied to synthesise a solution to the problem, if a solution exists.

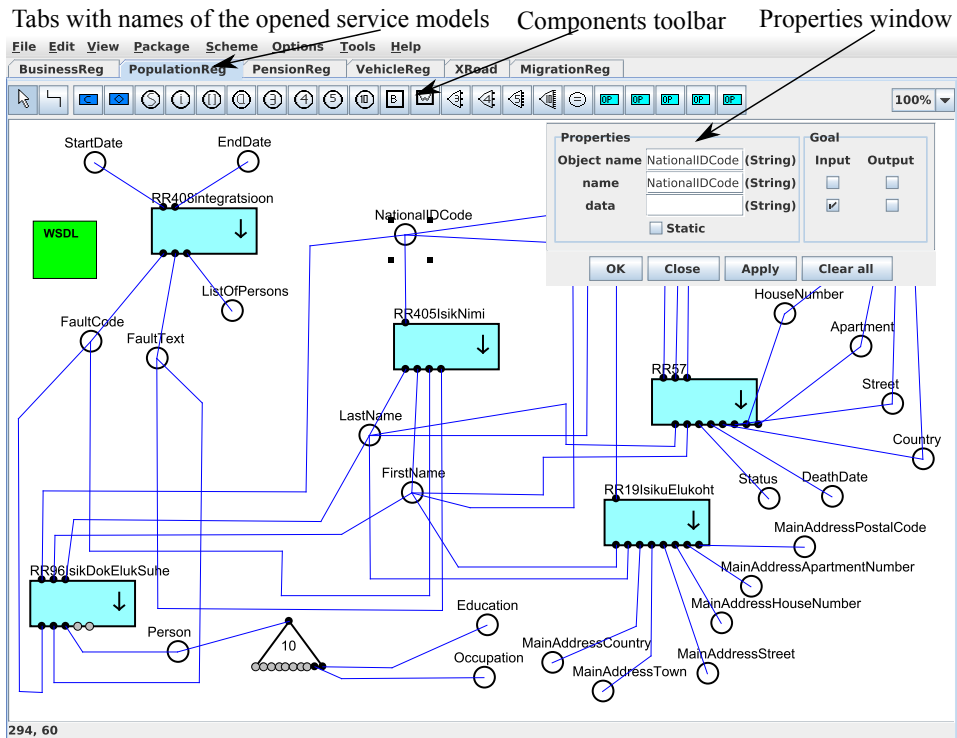


Figure 4.4. CoCoViLa Scheme Editor with properties window

When creating a package for a new domain, it is possible to use language components from other packages. This allows to create a software package that provides general purpose workflow description as a base language and extend this base language with domain specific components, for each possible domain. For example, different languages can be created for the composition front-end for

the e-government web services and for web services offering access to scientific data.

CoCoViLa includes two editors – Scheme Editor and Class Editor. Class Editor is used to create and modify software packages, including visual parts and component specifications. Class Editor allows to develop new components from scratch or reuse and modify components from existing software packages.

In the context of web service composition, Scheme Editor is used to create, visualise, modify, compile and run schemes, that is, to handle service models. It is possible to specify pre- or postconditions of the web service to be composed, using a properties window of the data entity and setting data to be either input or output (see Figure 4.4). It is possible to open more than one model in the same window. Names of opened models are shown in tabs on top of the window. Figure 4.4 shows the Scheme Editor window with four opened service models – XRoad, BusinessReg, VehicleReg, PopulationReg – and a properties window of the selected data entity – NationalIDCode. PopulationReg model’s tab is selected and its visual model is shown in the main window. Circles represent data entities, rectangles represent atomic web services and a square with a label “WSDL” represents a superclass that contains information that is common to the whole model. Components that are available for service models in population registry package are shown in components toolbar.

All visual components in CoCoViLa have a *metaclass* associated with them. Metaclass contains two parts: a logical specification part and a program part. The specification part is also called *metainterface* and it allows to use the logic described in Chapter 3, by using the specification language that is covered in Section 4.3.2. The specification is included in the metaclass as a comment between `/*@` and `@*/`. The program part includes the realisations for axioms given in the specification part and it is implemented as a set of Java methods. The specification language supports the inheritance of metaclasses. The following code shows a metaclass for the web service *GetAddress*.

```
class GetAddress extends Service {
    /*@ specification getAddress super Service {
        String firstName, lastName;
        String address;
        opName, inputMsg, outputMsg, firstName,
            lastName -> address {getAddress};
    }@*/
    public String getAddress (String opName, String inputMsg,
        String outputMsg, firstName, lastName) {
        String address;
        ...
        return address;
    }
}
```


The exact implementation of the realisation varies for different web services. Metaclass *GetAddress* inherits the variables *opName*, *inputMsg* and *outputMsg* from the metaclass *Service*. Other atomic web services also inherit common variables from the generic *Service* class. Metaclass *Service* is implemented as follows:

```
class Service {
    /*@ specification Service {
        String opName;
        String inputMsg;
        String outputMsg;
    }@*/
}
```

CoCoViLa is implemented in Java and therefore it also supports Java inheritance. This means that in addition to metainterface inheritance it is possible to inherit realisations. The specification language is covered in more detail in Subsection 4.3.2.

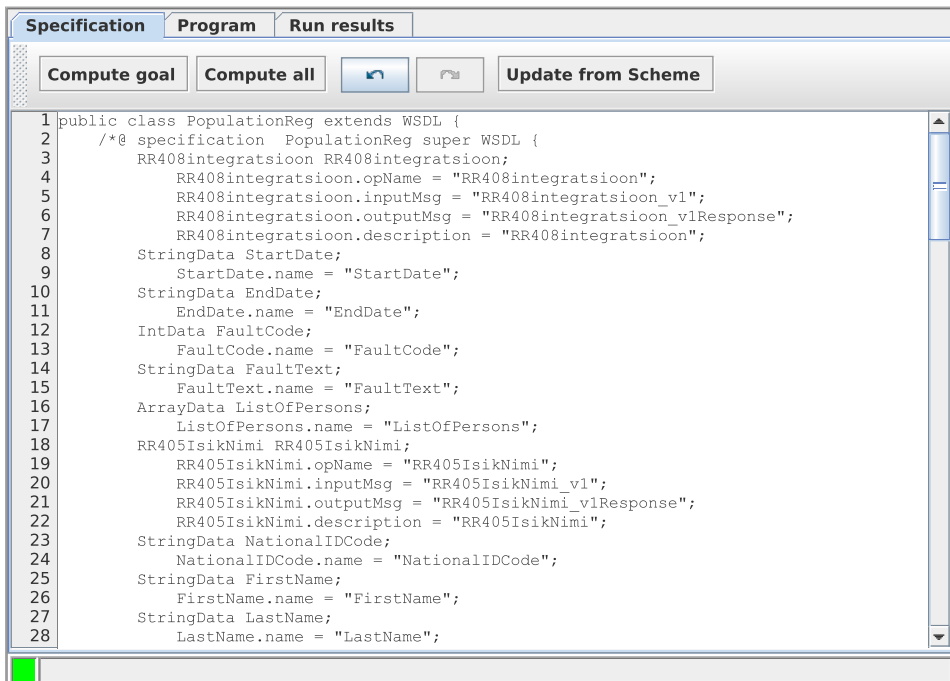


Figure 4.5. CoCoViLa specification window

Visual domain models have one-to-one translation to the model's textual form. Textual representation is a bit more expressive and allows to express, for example, logical pre-and postconditions and inheritance that might not be shown on visual

models. Textual specification of the visual model consists only of a metainterface that is used for synthesis. If the problem is solvable, then the realisations of scheme components are used for generating a program that solves the problem. This means that in case of very large models, or if no visualisation is necessary, visual part can be ignored and textual specification can be used instead. A fragment of the model's textual form is shown in the specification window in Figure 4.5. The specification extends the WSDL class that was set as a superclass in the Figure 4.4. Inheritance from the superclass is generated automatically into the textual specifications. If the goal is given and the scheme is available, SSP-based program synthesis method is applied to prove the computability of the goal on the given model. If the proof is found it is automatically transformed into a Java program. After compiling and running a Java program, the result is obtained. Scheme Editor supports usage of hierarchical models that allows to divide the complexity of large models into different layers.

4.3 Implementation of the Web Service Composition Tool

This section describes the implementation details of the web service composition tool. The method of automated web service composition has already been described in Chapter 3 and the tool's knowledge architecture was given in Section 4.1. Automatic steps made by the composition tool are shown in Figure 4.6. Note that this figure is almost the same as Figure 3.8, except, it has an additional input – superclass and an additional step where Java code is generated. This step is specific to the tool used – CoCoViLa, and not forced by the method of the web service composition proposed in this work.

Superclass allows to specify additional information about the web service to be composed. For example, a name of the resulting web service and a file where the output is written to can be defined by the properties of the superclass. Superclass also has an ability to gather data from the components of the service model. The result of the Java program depends on the superclass.

To use the composition tool, the developer of a new compound web service has to know how to specify the goal, that is, requested postconditions and needed preconditions of the compound web service. After the pre- and postconditions have been defined, a web service composition algorithm will be synthesised automatically. The structure of this algorithm already represents the structure of the compound web service to be generated. However, CoCoViLa produces only a Java code, but the user needs a representation in some process language (e.g., BPEL). Therefore the code uses preprogrammed generators of BPEL or OWL-S (which one is generated is defined by the superclass). When the code is executed, a web service description corresponding to the structure of the synthesised algorithm is generated using additional information from the initial specification.

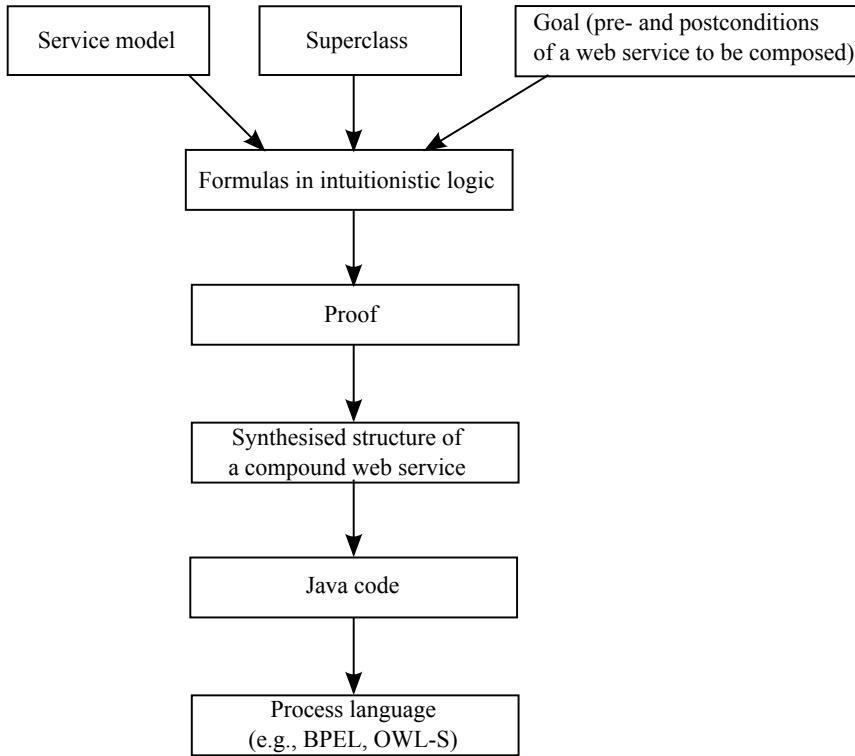


Figure 4.6. Automatic steps in the composition tool

All intermediate steps, that is, steps between defining a goal on the model and getting a compound web service description as an output, are done automatically by the composition tool. However, if necessary, intermediate steps (e.g., a structure of the compound web service or a generated Java code) can be visualised for the developer, for instance, for debugging purposes.

4.3.1 Composition Packages

A package is a collection of components and schemes related to an application domain. A package is described by a package description file in XML format. Each package can have its own visual language, but it is also possible to reuse components from other packages.

4.3.2 Specification Language for Web Service Composition

A specification language for the web service composition is described here. This language allows to specify the expressions of the logic that were described in Chapter 3 and it is implemented in CoCoViLa.

The specification language used for web service composition is the following:

- Specification of variables: `type id, [id2, ...];`
- Variable binding: `variable1 = variable2;` Variable binding specifies the equality between variables `variable1` and `variable2`.
- Axioms: `input -> output {realisation};` Axiom is a computability statement (see the Chapter 3 on logic), where `input` is a parameter for the method `realisation` and `output` is what is returned. Axiom's input may include multiple variables, separated by commas. Commas denote the conjunctions in logic and variables correspond to the propositional variables of intuitionistic logic. Axiom's output can be an array or a structure. Input may also include subtasks, for instance:
`[a->b], c -> output{realisation};`
 In this example, axiom's input contains a subtask `[a->b]`, and a variable `c`.
- Aliases:
`alias aliasName=(variableName1, ..., variableNameN);`
 Alias variable allows to define a structures containing several variables. It is useful in *Selector/Constructor* components that enable to take individual values out of the structure or to create structures.
- Wildcards: `alias aliasName=(*.variableName);`
 In the case of wildcards, alias structure will contain all the variables having the name `variableName`. Variables in alias with a wildcard depend on particular specification where such a statement occurs.
- Control variables: `void controlVariableName;`
 Control variables are used as logical pre- and postconditions.
- Specification language supports inheritance. Inheritance is used in two ways in the web service composition context:
 1. Specific atomic web service components inherit common variables and axioms from the generic *Service* component.
 2. A scheme can inherit from its superclass. This allows to direct the composition flow with control variables, pre- and postconditions and to gather data from scheme using wildcards.

The full syntax of the specification language used for web service composition is the following:

```

MetaInterface ::= 'specification' SpecClassName [InheritanceDecl]
                '{'Specification '}'
InheritanceDecl ::= 'super' SuperClassList
SuperClassList ::= SpecClassName [',' SuperClassList]
Specification ::= Statement ';' [Specification]
Statement ::= VariableDecl | Binding | Valuation | Axiom | Alias
VariableDecl ::= Type IdentList
IdentList ::= Identifier [',' IdentList]
Binding ::= Variable '=' Variable

```

```

Variable ::= Identifier | Variable '.' Identifier
Valuation ::= Variable '=' Value
Axiom ::= (SimpleAxiom | SubtaskAxiom) '{' Realization '}'
SimpleAxiom ::= VariableList '->' Variable
VariableList ::= Variable '[' ',' VariableList
SubtaskAxiom ::= SubtaskList '[' ',' VariableList ] '->' Variable
SubtaskList ::= Subtask '[' ',' SubtaskList
Subtask ::= '[' VariableList '->' VariableList ']'
Goal ::= [GoalInputs] '->' GoalOutputs
GoalInputs ::= VariableList
GoalOutputs ::= VariableList
Alias ::= (AliasDeclaration ['=' AliasStructure]) | AliasDefinition
AliasDeclaration ::= 'alias' '[' (' Type') Identifier
AliasStructure ::= '(' ( VariableList | AliasWildcard ) ')'
AliasWildcard ::= '*' Identifier
AliasDefinition ::= Identifier '=' '[' VariableList ']'
Type ::= JavaType | SpecClassName | 'void' | 'any'
JavaType ::= JavaReferenceType | PrimitiveType
Identifier ::= Letter (Letter | Number | '_' ) *

```

In addition to these language constructs, CoCoViLa also supports equations and exceptions that have not been used for web service composition in this thesis. Full description of the specification language of the tool used can be found from the documentation on CoCoViLa's homepage [5].

4.3.3 Visual Language for Web Service Composition

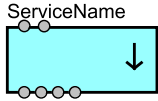
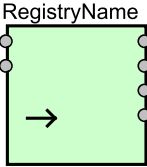
CoCoViLa allows the creation of domain specific visual languages. In this work, the implemented language represents web services, control nodes, data entities, transformation components and their connections. It is used to create/generate and add data to schemes, that is, visual representations of service models. There are three types of components in the visual language for web service composition: components that are used in submodels (web service components), components that are used in upper level hierarchical model (registry components) and general components available to all models. Inputs and outputs in the visual language are represented by small circles, called ports and specification variables accessible through the properties window are called fields.

Generic *atomic web service* and *registry* components are shown in Table 4.1. General components are described in Table 4.2. Note that the number of ports and fields of components representing atomic web services and registry components can be different and depends on actual web services, but in general they look like shown in Table 4.1.

4.3.4 Output Generation

As it was stated earlier, a form of the output depends on the superclass that is given. There are two ways to generate output:

Table 4.1. Atomic web service and registry components of the visual language




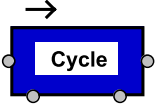

Visual image	Meaning	Description
	Atomic web service	<p><i>Atomic web service</i> allows to specify pre- and postconditions by connecting respective ports to the data entities. Which ports represent pre- and which ones are for postconditions is indicated by the arrow. In addition, it is possible to specify service name, location, messages, etc. in the properties window of the component. Different atomic web services can have different number of ports and these ports can have different types. But their general look is like shown on the figure.</p>
	Registry	<p><i>Registry</i> components enable to use submodel's information from the upper level model. They represent web services from one registry or information system. Different registry components can have different number of ports.</p>

1. To instantiate all components with a correct text that is written to a file when the component is reached. This approach is good when simulating the execution. This approach also allows to create realisations of web services that do not output text but invoke the web service. Text or other preconditions can be set from the superclass by using the wildcard. This, however, does not work well with block-structured languages like BPEL, where the process contains information about all the variables involved, followed by the web services and control nodes involved and where text needs to be generated both when a control node is entered and when it is left.
2. To extract the structure of the compound web service first and to generate the output from the superclass. This approach works well when block-structured output language needs to be generated.

The second approach has been used in the experiments on service models described in this thesis. Previously, experiments have also been done as a part

of this work with generating text from the components that were a part of simple workflows [37].

Table 4.2. General components of the visual language

Visual image	Meaning	Description
DataEntityName 	Data entity	<i>Data entity</i> has a hidden port in its center. Data entities from different types also have ports of different types.
	Selector/ Constructor	<i>Selector/Constructor</i> works both ways, it either takes elements out of the structure or inserts elements into the structure. Different components exist for structures with different number of elements.
	Same as	<i>Same as</i> component is used to connect data entities that have different names but the same meaning.
	Cycle	<i>Cycle</i> is a higher-order component that is used to cycle through a subtask specified by its pre-and postconditions connected by the ports at the bottom of the Cycle component. (Other higher-order components are possible.)
	Superclass	<i>Superclass</i> is used to insert new information (process name, description, etc.) and to choose an output format. In this case, output format is going to be BPEL, but other formats are possible.

4.4 Conclusion

Knowledge architecture and implementation details of the web service composition tool, developed in this work, was described in this thesis. This tool enables the user to work with visual service models. At the same time it is able to handle service models automatically to reason about the goals, that is, desired compound web services, given by the user. In order to see if the tool is scalable and applicable to real world web services, experiments were done on Estonian e-government web services. These experiments are described in Chapters 5 and 6.

5 Web Service Composition on Large Service Models

In Chapter 4 a web service composition tool, developed as part of this work, was described. This chapter describes the experiments done in order to test the tools applicability to large real-world service models. X-Road service descriptions are used for experiments described in this chapter and the following chapter. The experiments on service models that are part of this work started in 2007. At this time X-Road [20, 21] contained almost 1000 web services. Today the number of web services is more than 2000 [49]. Experiments on the X-Road service mode which was created by Peep K ungas [27] in 2006 are described in this chapter. Experiments on hierarchical X-Road service models are described in Chapter 6.

This chapter starts with a description of the initial X-Road service model, followed by its equivalent in CoCoViLa. After the description of the service model the synthesis of compound web services on this model is explained on an example.

5.1 X-Road Model

The first X-Road model that was used in this work, was a syntactic service model created by Peep K ungas in 2006 [27], it included about 300 atomic web services and about 600 unique references to data entities. This model was a graph in GML (Graphical Modelling Language) format. Figure 5.1 shows a visual representation of the whole service model visualised by yEd graph editor [70]. It is a large graph where nodes are atomic web services and data entities.

A small part of the model shown by a rectangle in Figure 5.1 is enlarged in the Figure 5.2. List in the left pane of the user interface fragment shows a scrollable list of all components. Rectangles represent atomic web services and circles represent inputs and outputs that are connected to web services in this model fragment. Size of the circle shows its relative importance (connectivity to web services). A resource with the largest value in this model is *NationalIdCode* that is used in most of the web services in Estonian e-government information system.

The model presented here was the basis for the automatic synthesis of web services in the experiments on large service models. However, it had to be transformed into CoCoViLa format in order to be applicable as a specification for the synthesis.

5.2 Service Model in CoCoViLa

To use the model described in the previous section in CoCoViLa – a program was created that automatically translated models in GML format into CoCoViLa models. Translation was quite easy, as the initial model only contained information about the names of atomic web services, data entities and the input/output dependencies. All atomic web services were created as components of one package, inputs and outputs of atomic web services were considered to have a type *String* or a structure consisting of *Strings*. Therefore, all data entities were generated with a type *String* or as tuples of *Strings*. The initial model did not contain any information about the actual messages exchanged, thus, all the input messages were generated in the form *get* + *<servicename>* and output messages were generated in the form *<servicename>*+ *Response*.

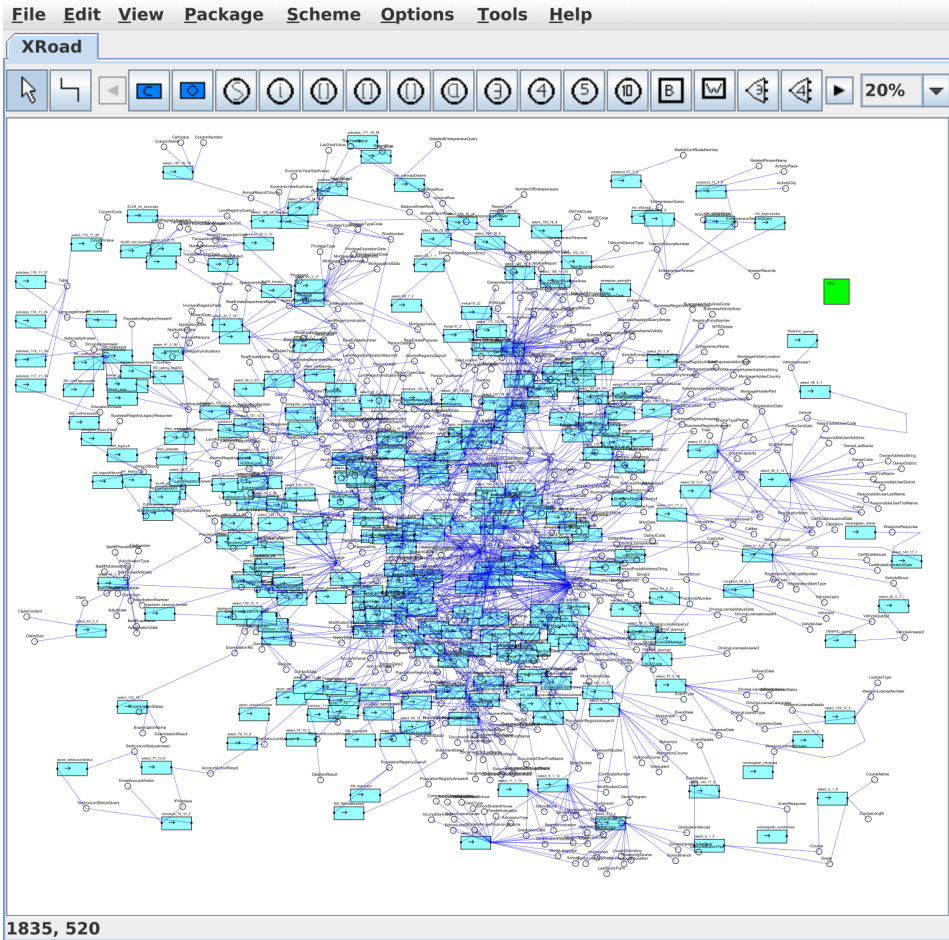


Figure 5.3. X-Road model in CoCoViLa

The service model represented in CoCoViLa is shown in Figure 5.3. This model can be zoomed in or out as needed. All components in this model are part of one package in CoCoViLa, therefore, the number of components in the toolbar is quite large. As in the initial model, rectangles represent atomic web services and circles represent data entities. Note that the visual notation used in this thesis for large service models is slightly different from the notation used in the earlier published papers. Notation in the papers was more compact which was important for large models. The notation used in the thesis is different in order to conform to the notation developed for hierarchical service composition.

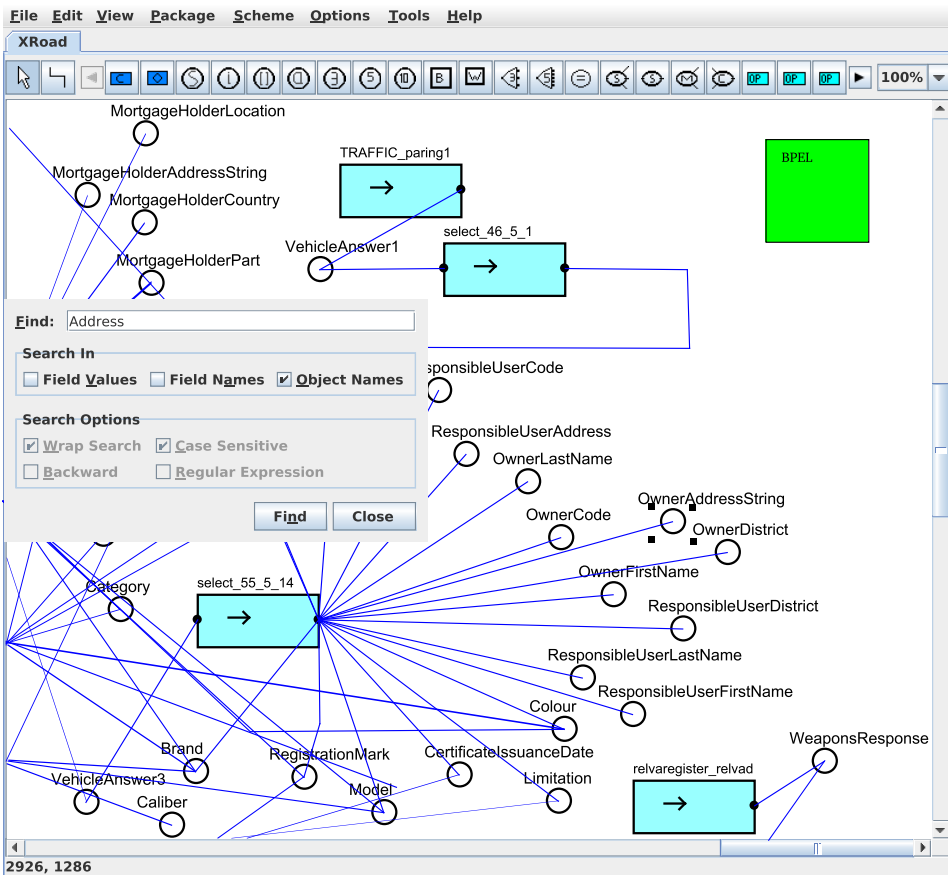


Figure 5.4. Zoomed in X-Road model with a search window in CoCoViLa

By opening the data entity's properties window it is possible to define known preconditions and desired postconditions from which the goal web service is formulated. To simplify this task on such a large model, CoCoViLa offers a possibility to search on a scheme (i.e., a visual representation of the model). It is possible to search either by component's (object in CoCoViLa) name or by the

name or a value of the field. If a component is found it is focused and highlighted on the scheme. This is shown in Figure 5.4, which represents a fragment of the zoomed in X-Road service model in CoCoViLa. This figure also shows a search window with a search string *Address* and one of the highlighted data entities that is returned as a result – *OwnerAddressString*.

5.3 Automatic Handling of a Service Model

In order to illustrate the composition process of the web service composition tool described in Chapter 4, an example of composition on a large service model described in the previous section is given here. Consider, for instance, that an official has to find a person's occupation and a contact address, having only information about the person's graduation certificate. When finding the information, official has to keep in mind that different databases of X-Road registries have their own copy of person's address, which may not be the same in all registries. Therefore, to find all possible addresses of a particular person, an official has to query all databases that may have information about addresses. Currently these queries have to be executed separately on different databases. A compound web service would allow inserting a graduation certificate's number or another identifier as an input to get an occupation area and different contact addresses as an output.

To create this compound web service in CoCoViLa, the user must, first, to define the output format by selecting corresponding superclass and to specify other relevant details, for example, the name of the output process and the file where the output is written. These details depend on the superclass that has been selected. In this example, BPEL superclass is selected. This is shown in Figure 5.3. The second step is to find relevant data entities on the service model. These data entities are *GraduationCertificate* as a precondition and *OccupationArea*, *AddressString*, *EstonianAddressString*, *OwnerAddressString*, *ResidencyAddress* as postconditions. Search window can be used to find relevant data entities, like it has been shown in Figure 5.3, where data entities representing address are looked up. By marking these variables as pre- or postconditions in the model or specification, and by selecting a superclass, the user gives a goal for the synthesiser.

After a goal is defined by the user, the steps shown in Figure 4.6 are performed. That is, a service model and the goal are automatically translated into a textual specification, which is used to generate a Java program if the proof of solvability of the problem can be built. A part of the textual specification is shown in Figure 5.5. The last line in this figure shows the goal that was specified by the user on the visual model:

```
GraduationCertificate.data -> ResidencyAddress.data,  
AddressString.data, OwnerAddressString.data,  
OccupationArea.data, EstonianAddressString.data;
```

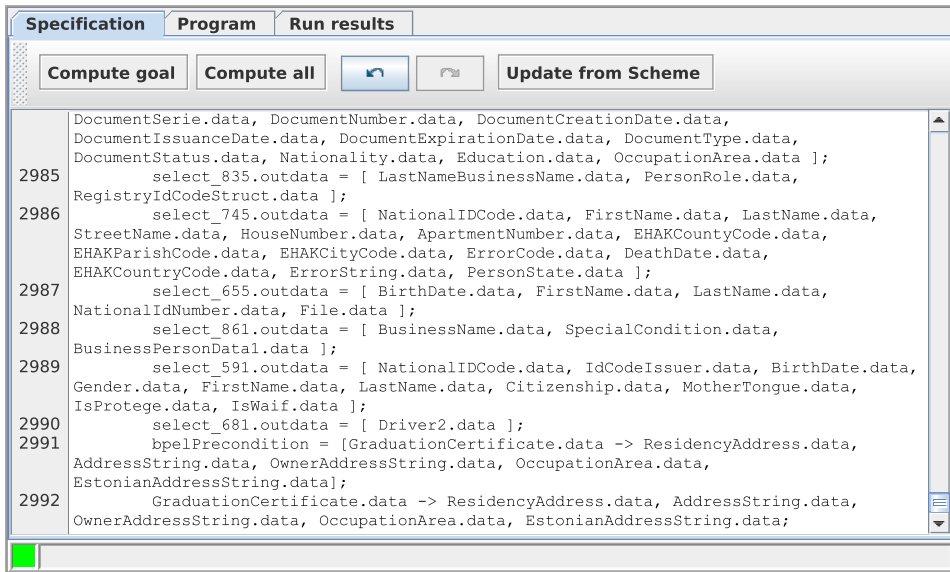


Figure 5.5. Service model's textual specification window

An application window for viewing generated Java programs is shown in Figure 5.6. The last line of the program in this figure shows a method *createProcess* that is called from the BPEL superclass when the structure of the goal web service has been synthesised and the results are gathered into *alias* called *bpelPrecondition* that is a precondition to the generation of BPEL output:

```

createProcess(alias_bpelPrecondition_35719, processName,
processComment, processNamespace, outputFile);

```

The other parameters for the *createProcess* method are given in the superclass component's properties window. BPEL description of the compound web service is generated by running the Java program. If the structure of the compound service cannot be synthesised, BPEL output is not generated as the precondition *bpelPrecondition* is not satisfied.

As the result of this example, eleven different atomic web services have to be invoked in order to solve the task specified by the goal. When BPEL superclass is selected the result is a process description in BPEL, that is, an XML file with BPEL process. Because the initial model did not contain all the necessary information, many details have been omitted from the BPEL description generated by the CoCoViLa composition tool. For example, there is no information about *messageTypes* and service names. Despite many omissions, the resulting web service description is 118 lines long. Figure 5.7 shows a fragment of the BPEL description.

```

1032     PersonStatus.data = (String)alias_select_720_outdata_26543[4];
1033     RelationType.data = (String)alias_select_720_outdata_26543[5];
1034     FatherName.data = (String)alias_select_720_outdata_26543[6];
1035     String[] alias_select_739_outdata_26465 = select_739.select(select_739.opName,
select_739.indata, select_739.output);
1036     NationalIDCode.data = (String)alias_select_739_outdata_26465[0];
1037     FirstName.data = (String)alias_select_739_outdata_26465[1];
1038     LastName.data = (String)alias_select_739_outdata_26465[2];
1039     PersonStatus.data = (String)alias_select_739_outdata_26465[3];
1040     RelationType.data = (String)alias_select_739_outdata_26465[4];
1041     FatherName.data = (String)alias_select_739_outdata_26465[5];
1042     EstonianAddressString.data = (String)alias_select_739_outdata_26465[6];
1043     BirthPlace.data = (String)alias_select_739_outdata_26465[7];
1044     RelationStatus.data = (String)alias_select_739_outdata_26465[8];
1045     Object[] alias_bpelPrecondition_27129 =new Object[5];
1046     alias_bpelPrecondition_27129[0] = ResidencyAddress.data;
1047     alias_bpelPrecondition_27129[1] = AddressString.data;
1048     alias_bpelPrecondition_27129[2] = OwnerAddressString.data;
1049     alias_bpelPrecondition_27129[3] = OccupationArea.data;
1050     alias_bpelPrecondition_27129[4] = EstonianAddressString.data;
1051     createProcess(alias_bpelPrecondition_27129, processName, processComment,
processNamespace, outputFile);

```

Figure 5.6. Program window showing the resulting Java program

A BPEL description starts with the definition of the process, it proceeds with defining variables used and then a structure of the process is described. In this case, it is a sequence containing invocations of atomic web services. The `<assign>` part takes care of copying the values of data entities to the relevant messages. Figure 5.8 represents the resulting BPEL process visualised in Eclipse BPEL editor [11].

Any service model can also contain higher-order control nodes that can be added by the user and saved as part of the service model. The usage of higher-order nodes is demonstrated in Chapter 6.

The synthesis algorithm without higher-order nodes has linear time complexity and can be applied to very large service models. The time spent for solving the example explained in this chapter was about 2.5 seconds on a laptop with 1.2 GHz Intel processor. This includes specification parsing, planning and code generation time.

```

<?xml version="1.0" encoding="UTF-8"?>
<process name="X-road compound service"
targetNamespace="http://x-tee.riik.ee/example_compound_service"
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
...>
<documentation xml:lang="EN">X-road compound service
</documentation>
<variables>
<variable name="Request" messageType="getComplexServiceRequest"/>
<variable name="getselect_4_1_5"/>
<variable name="select_4_1_5Response"/>
<variable name="getehis_kod_oppur_laps"/>
<variable name="ehis_kod_oppur_lapsResponse"/>
<variable name="getTRAFFIC_paring22"/>
<variable name="TRAFFIC_paring22Response"/>
...
</variables>
<sequence>
<receive createInstance="yes" name="start"
operation="getComplexService" variable="Request"/>
<invoke name="select_4_1_5"
operation="select_4_1_5"
inputVariable="getselect_4_1_5"
outputVariable="select_4_1_5Response"/>
<assign>
<copy>
<from variable="select_4_1_5Response" part="NationalIdCode"/>
<to variable="getehis_kod_oppur_laps"/>
</copy>
</assign>
<invoke name="ehis_kod_oppur_laps"
operation="ehis_kod_oppur_laps"
inputVariable="getehis_kod_oppur_laps"
outputVariable="ehis_kod_oppur_lapsResponse"/>
<assign>
<copy>
<from variable="select_4_1_5Response" part="NationalIdCode"/>
<to variable="getRR_RR40isikTaielikIsikukood"/>
</copy>
</assign>
<invoke name="RR_RR40isikTaielikIsikukood"
operation="RR_RR40isikTaielikIsikukood"
inputVariable="getRR_RR40isikTaielikIsikukood"
outputVariable="RR_RR40isikTaielikIsikukoodResponse"/>
...
</sequence>
</process>

```

Figure 5.7. BPEL output in XML

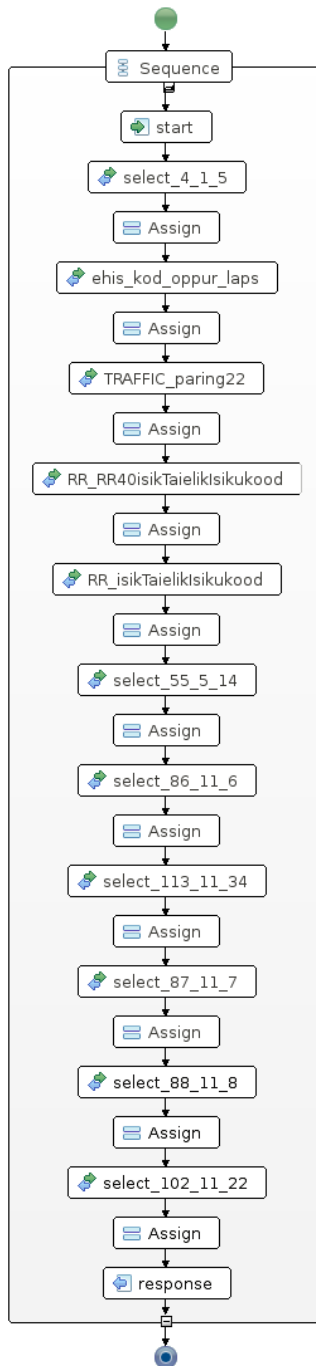


Figure 5.8. The resulting compound web service visualised in Eclipse BPEL editor

5.4 Conclusion

It was demonstrated in this chapter that the web service composition method, proposed in this thesis, is applicable to very large service models. Planning on models of this size works fast, however, service models of this size are rather hard use manually. Because of this, complexity of the service model needed to be reduced. This led to the development of hierarchical service models. In order to create models as realistic as possible and to simplify service model generation, programs were created to generate service models from real Estonian e-government service descriptions. The hierarchical X-Road model and the experiments on this model are covered in Chapter 6.

6 Web Service Composition on Hierarchical Service Models

Although the initial service model of X-Road described in Chapter 5 was a good reflection of reality, it was lacking some important details, like information about the names and types of input and output messages and exact names and types of data entities. Even the information about the provider was not explicit. In addition, it was one large flat model with many components (almost 900). Therefore, it was hard for the end user to use and modify the model's visual representation or define a goal. Definition of a goal required a knowledge of a large joint ontology of all services of all ministries and organisations that have joined X-Road. This, of course, is difficult for users. These were the reasons that led to the experiments with hierarchical service models, generated from SAWSDL descriptions. These experiments are described in this chapter. The chapter starts with a description of hierarchical service model implementation details. After that a few words are said about automatic generation of CoCoViLa packages and service models. Finally, examples of composition problems are described on the hierarchical X-Road service model.

6.1 Hierarchical X-Road Service Model

It was said already in Chapter 2 that components on the upper level of the hierarchical service model correspond to submodels that represent atomic web services from one or more web service providers that share a common ontology. In the X-Road service model, one component of the hierarchical service model represents the web services from the registry or database of one ministry or organisation.

For composition on hierarchical models, a different package was created for each provider and for the upper layer X-Road model. The buttons on the toolbar represent only the components that are relevant to the corresponding submodel. This allows the user to work with a smaller number of known concepts. All packages include also some common general components, for example, a cycle, a condition, components to represent data entities, superclasses and common data transformation components. The main difference between these packages comes from atomic web service components that have different pre- and postconditions. All web service components inherit from the generic *Service* metaclass. A package can contain more than one service model. This allows to create different views from the web services provided by one organisation.

6.1.1 Generation of a Service Model in CoCoViLa

Automated generation of a hierarchical service model was already investigated in Section 2.3 of Chapter 2. Details specific to CoCoViLa service model generation are given in this subsection.

Existing partially annotated SAWSDL descriptions were used to generate sub-models. It should be mentioned that annotating X-Road web service descriptions was not a part of this thesis. The SAWSDL extension of EasyWSDL Toolbox [10] was used to parse SAWSDL descriptions from the Java program. The program then generated a package description in XML, scheme of the service model and metaclasses for web service components. Components, corresponding to the sub-models, were generated for the X-Road package that is used for defining upper level views of hierarchical service models.

Some components are shared among different packages, for example, components for data entities of different types, control constructs, selector/constructor components, BPEL and WSDL superclasses. When a new package is generated, description of these components is added to the package XML description, using the XML external entity references. However, information about these components still had to be generated into the service model description, using the information extracted from the SAWSDL description.

X-Road web services currently use WSDL 1.1. WSDL specification allows slight variations in its format, but, in general, WSDL description is similar to the example *WeatherService* shown in Figure 6.1. More information on WSDL can be found in WSDL 1.1 specification [66]. Requirements for X-Road services are specified by the *Requirements on information systems and adapter servers* document [68]. WSDL description usually contains a section for *types* that includes definitions for *simpleTypes* and *complexTypees*. It also includes *messages* that can be sent to *operations*. Messages can include *parts*. Abstract operations are defined in the *portType*. *Binding* section defines message format and protocol details for operations and messages defined by a particular *portType*. Binding details are omitted from the example, because they do not show up on the visual service model representation and in examples about X-Road web services. Execution details are not used since these web services cannot be accessed without correct access rights. SAWSDL descriptions are used to generate service models in this work. SAWSDL description is a WSDL description with semantic annotations. Semantic annotations added to elements are shown in Figure 6.1 using a *modelReference*, for example:

```
sawsdl:modelReference="http://<ontology>#TownName"
```

where *<ontology>* refers to the URL of the ontology used and *TownName* is the concept from the ontology.

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions>
  <types>
    <complexType name="locationType">
      <all>
        <element name="town"
          sawsdl:modelReference="http://<ontology>#TownName"
          type="xsd:string" />
        <element name="zipcode"
          sawsdl:modelReference="http://<ontology>#ZipCode"
          type="xsd:string"/>
      </all>
    </complexType>
  </types>
  <message name="GetWeatherInput">
    <part name="location" type="locationType">
  </message>
  <message name="GetWeatherOutput">
    <part name="temp"
      sawsdl:modelReference="http://<ontology>#TemperatureInCelsius"
      type="xsd:string" />
  </message>
  <portType name="WeatherServicePort">
    <operation name="GetWeather">
      <input message="GetWeatherInput"/>
      <output message="getWeatherOutput"/>
    </operation>
  </portType>
  <binding name="WeatherServiceBinding">
    ...
  </binding>
  <service name="WeatherService">
    <port name="WeatherServicePort"
      binding="WeatherServiceBinding">
    </port>
  </service>
</definitions>

```

Figure 6.1. Simplified example of the SAWSDL description for the *WeatherService*

Data Entities

In the implementation of the web service composition tool the general part of the service composition package description includes components for specifying data entities. Different data entity components have different port types. For example, data entity components exist for *xsd:int* type (port type *int*) and *xsd:string* type (port type *String*). *ComplexTypes* are represented as structured data entities that have a port type *alias*. Elements in SAWSDL *types* section are translated into data entities. Because data entities are already defined in the default composition

package, they only need to be generated into the description of the service model. Name of the data entity object in the service model will be the concept extracted from the semantical annotation. For the example in Figure 6.1 data entities with names *TownName* and *ZipCode* and with port types *String* are generated to the service model.

Because data entities are generated from the concepts of elements defined in the types section of the SAWSDL description, they depend on the ontology used for the service model. Although the web service composition tool allows to model very complex data types, automated generation is implemented for *simpleTypes* and for *complexType*s that are built from XML Schema simple types and represent types of input or output messages. Data entities for arrays and for *complexType*s that contain other *complexType*s have to be added manually using the data entities with type *alias* and *Selector/Constructor* components. It is possible to automate the generation of all data entities, but it needs deeper analysis and probably also restructuring of the X-Road WSDL descriptions (see the discussion in Section 2.3.2). This, however, is out of the scope of this thesis.

Atomic Web Services

Atomic web services are generated from the WSDL operations. Because different operations have different pre- and postconditions, these components are generated as different atomic web services that extend the generic *Service* component. In case of the example from Figure 6.1, an atomic web service with a name *GetWeather*, with input message *GetWeatherInput* and output message *GetWeatherOutput* are generated. Pre- and postconditions for planning are extracted from the parts of messages or from their types and they represent the actual variables not the concepts from the ontology. In this example the variables *town* and *zipcode* become preconditions and *temp* becomes a postcondition for the web service *GetWeather*. In logical specification this web service is specified as follows:

```
town, zipcode -> temp {GetWeather};
```

Using real variable names is necessary, because this is the data used to invoke atomic web services and therefore this information needs to be generated also into the compound web service description.

Connections

Automatically created connections are added to the service model. Connections are created between data entities and atomic web service pre- and postconditions with the same concept name. Variables with different names are automatically connected if they are marked with the same concept. An expert user can later modify the service model and add data dependency relations to connect components with different names or types.

WSDL Superclass

WSDL superclass contains information about the WSDL description. For example, name and location of the description file itself and name and the location of the (WSDL) service(s) described in WSDL. In case of the example in Figure 6.1 the name of the service generated to the WSDL component description would be *WeatherService*. Correct values are added to the service model's WSDL superclass object for each submodel.

6.1.2 Submodels

The hierarchical X-Road service model that is used for examples in this chapter is created from five annotated WSDL descriptions, describing atomic web services offered by business registry, population registry, vehicle registry, migration registry and pension insurance registry.

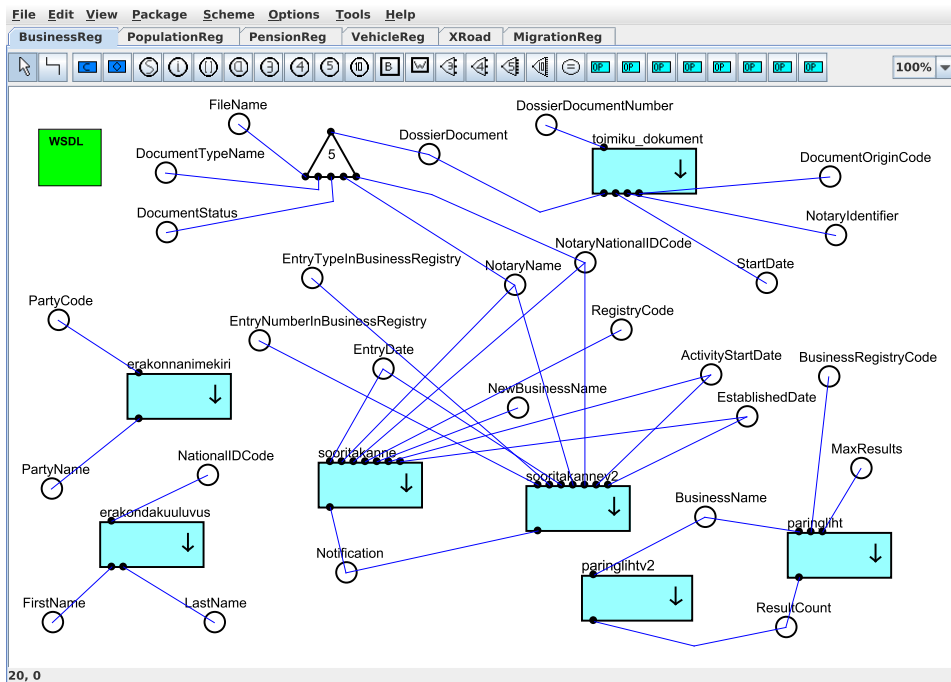


Figure 6.2. Fragment of the business registry's service model

Figure 6.2 shows a fragment of the business registry service model. Rectangles represent web services and circles represent data entities in this figure. Arrows on the rectangles represent which ports are preconditions and which ports are postconditions for the particular atomic web service. WSDL component, declared as a superclass, in the top left corner contains additional information extracted

from the web service description about the particular registry. For readability, only 8 atomic web services out of the 42 available for the business registry are shown in this model. Service models for other registries are similar. Submodels can include higher-order components that are hidden in the visual representation of the upper layer, but are used in the synthesis.

6.1.3 Hierarchical Model

All service models can be saved as hierarchical components and used to create hierarchical service models. When creating a hierarchical component, the developer can select which pre- and postconditions of the submodel are shown on the hierarchical component. A hierarchical model of X-Road could represent the whole X-Road, but in this case the number of hierarchical components would be more than 100 (at least one for each organisation and ministry that has joined X-Road) which would be rather hard to handle in a single model. In addition, X-Road has quite complex access controls. Although this work currently does not consider access rights, it is obvious that it is not useful to add components with very tight access restrictions to all models. Fortunately, it is not necessary to put all components into one model and it is possible to create different views for users with different expertise and access rights.

Figure 6.3 shows a hierarchical view of the X-Road model. Rectangles with pre- and postconditions (marked by the arrows) are hierarchical components that contain submodels. The submodel for the business registry component was shown in Figure 6.2. In addition, there are hierarchical components for migration registry, vehicle registry, population registry and pension insurance registry. Information in the submodels is used in the synthesis but the complexity of the submodels is hidden from the user. Some data entities that exist in the submodels are also hidden and only those data entities that might be interesting to the user of the upper layer model are brought out as ports for pre- and postconditions.

BPEL component in the top left corner in Figure 6.3 is a superclass for the scheme and contains additional information about the compound web service to be composed (e.g., its name and description) and the output file. This information has to be entered by the user and it constitutes preconditions for the axiom in the BPEL superclass that generates the BPEL output. In the synthesis process this component will also gather information about the web services included into the workflow.

White triangles in the Figure 6.3 are *Selector* components that take elements out of the tuple. For example, triangle marked with number 5, takes *Document-TypeName* and *NotaryNationalIDCode* out of the structure *BusinessDocument*. *NotaryNationalIDCode* is connected to the *NationalIDCode* through the *Sameas* component. Saying that two concepts with different names have the same meaning on the model requires the background knowledge from the expert, who can confirm that these data entities are in fact the same in this context. Higher-order

components can be added by the user. The usage of higher-order component *Cycle* is shown in the second example of the following section.

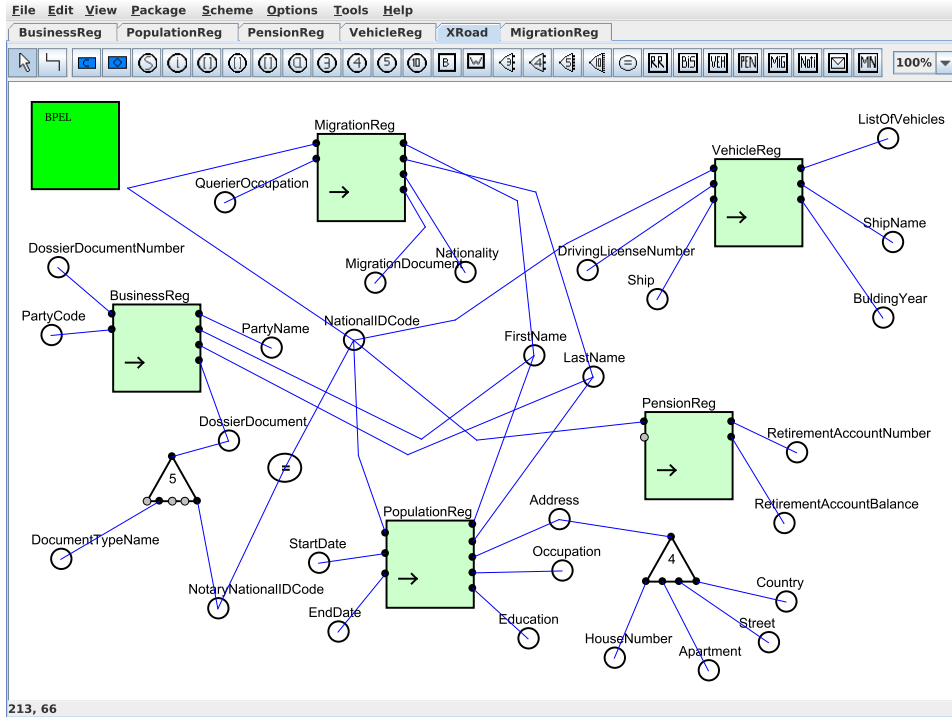


Figure 6.3. Hierarchical view of the X-Road model

6.2 Composition on Hierarchical Service Models

The composition process on a hierarchical service model is shown on examples in this section. In general everything works in the same way as in the example on large models (Chapter 5). The difference is that parts of the service model are hidden inside hierarchical components. The user is usually working on the upper layer of the service model, so this is where the superclass for generating output is chosen and where the goal, specifying the service to be composed, is defined.

Otherwise the steps for composition are exactly the same. The visual representation is translated into a textual specification that is used for proving the goal. If the goal is derivable from the service model, a program to generate the result is synthesised. All these steps, except the one where input is given to the user, can be hidden from the user.

6.2.1 Composition without Higher-Order Nodes

An example on a hierarchical service model that does not contain higher-order nodes is explained in this subsection. The hierarchical model shown in Figure 6.3 is used as the service model in this example.

The following example solves the problem where parts of notary's home address (*Street*, *HouseNumber*, *Apartment*) and information about the vehicles this notary owns (*ListOfVehicles*) needs to be discovered from a given document number (*DocumentNumber*). The developer of the service should know that person's home address can be obtained from person's *NationalIDCode*. From *DocumentNumber* it is possible to obtain *NotaryNationalIDCode* that can be connected to the *NationalIDCode* using the *Sameas* component to create the data dependency relation. This is also shown in Figure 6.3. The goal for constructing this compound web service on the given service model is the following:

```
DocumentNumber.data->ListOfVehicles.data, Street.data,  
HouseNumber.data, Apartment.data
```

The user does not need to know anything about the exact names of web services or even the registries that provide these web services. After setting the goal and running the tool, invocations of operations (i.e., atomic web services) are added to the resulting service in the following order:

```
arireg:toimiku_dokument (BusinessReg)  
autoregister:paring22 (VehicleReg)  
rr:RR405IsikNimi (PopulationReg)  
rr:RR57 (PopulationReg)
```

The name before the colon shows the service name extracted from WSDL description, saved into the variable of the WSDL superclass and propagated into atomic web services. The name after the colon indicates the name of the operation (atomic web service) and the text in the bold shows the registry, that is, a submodel, to which the web service belongs. Figure 6.4 shows the resulting web service visualised in Eclipse BPEL editor. Note that Eclipse BPEL editor visualises the WSDL service names instead of the operation names that are modelled as atomic web services in the service model. This is why there are two web services named *rr* on Figure 6.4. They represent two different atomic web services (i.e., operations) (*RR405IsikNimi* and *RR57*) of the same WSDL service.

Figure 6.5 shows a fragment of the BPEL output which contains 59 lines. XPath [69] expressions have been used for describing the *assign* statements synthesised from the X-Road model generated from SAWSDL descriptions. Using XPath expressions in *assign* statements allows to copy variables from complex data structures that are modelled using the *alias* data type and the *Selector* component.

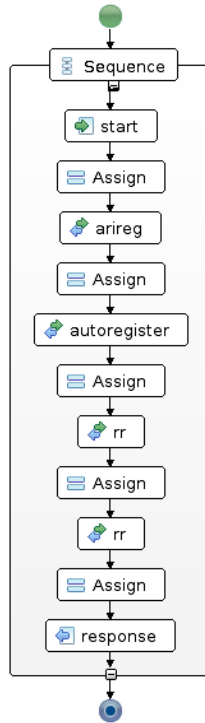


Figure 6.4. Result of the hierarchical composition visualised in Eclipse BPEL editor

The time spent for solving the example of composing four atomic web services without higher-order nodes was about 0.7 seconds on a laptop with 1.2 GHz Intel processor. This time included the specification parsing, planning and code generation time.

```

<?xml version="1.0" encoding="UTF-8"?>
<process name="X-road compound web service" ...>
  <variables>
    <variable name="Request"
      messageType="wsdl:getComplexServiceRequest"/>
    <variable name="toimiku_dokumentInputMessage"
      messageType="toimiku_dokumentInputMessage"/>
    <variable name="toimiku_dokumentOutputMessage"
      messageType="toimiku_dokumentOutputMessage"/ >
    ...
    <variable name="RR57_v1" messageType="RR57_v1"/>
    <variable name="RR57_v1Response" messageType="RR57_v1Response"/>
    <variable name="Response"
      messageType="wsdl:getComplexServiceResponse"/>
  </variables>
  <sequence>
    <receive createInstance="yes" name="start"
      operation="getComplexService"
      portType="wsdl:XRoadClientP" variable="Request"/>
    ...
    <invoke name="arireg" operation="toimiku_dokument"
      inputVariable="gettoimiku_dokument"
      outputVariable="toimiku_dokumentResponse"/>
    ...
    <assign>
      <copy>
        <from>${toimiku_dokumentInputMessage/notari_isikukood}</from>
        <to>${RR57_v1/Isikukood}</to>
      </copy>
      <copy>
        <from>${RR405IsikNimi_v1Response/Isikuenimi}</from>
        <to>${RR57_v1/Eesnimi}</to>
      </copy>
      <copy>
        <from>${RR405IsikNimi_v1Response/Isikupnimi}</from>
        <to>${RR57_v1/Perenimi}</to>
      </copy>
    </assign>
    <invoke name="rr" operation="RR57" inputVariable="getRR57"
      outputVariable="RR57Response"/>
    ...
    <reply name="response" operation="endComplexService"
      variable="Response"/>
  </sequence>
</process>

```

Figure 6.5. BPEL description for hierarchical composition without higher-order nodes

6.2.2 Composition with Higher-Order Nodes

This subsection demonstrates the usage of higher-order components on hierarchical service models. Higher-order components can be added both to the upper level models and to the submodels. Components added to the submodels are unfolded in the flat logical representation. This means that higher-order components on the submodels do not make the composition more complex than higher-order components on the upper level models.

Defining Services Models with Higher-Order Nodes

Ideally, submodels with higher-order components should be generated from web service descriptions that are able to represent higher-order components, for example, from BPEL descriptions. However, BPEL and other process description languages, such as, OWL-S, are meant to describe one workflow and not all workflows possible in the domain. Therefore, these languages are not the best choice for describing service models and currently higher-order components are added manually to the service models generated from SAWSDL descriptions.

One solution for description of a service model might be the creation of a service model description language that extends, for example, SAWSDL. This is one consideration for the future work. Another solution would be to create or use some implementation specific service model description language. At the moment, however, it is possible to save service models as CoCoViLa's scheme descriptions.

Example

The following example shows the usage of higher-order component *Cycle* that is added to the upper level model in order to cycle through the list of document numbers to discover the home addresses for the notaries related to the document. The service model used in this example is almost the same as the one used in the previous example (Figure 6.3), except it has an additional higher-order component *Cycle*. The modified service model is shown in Figure 6.6. The *Cycle* component has *ListOfDocumentNumbers* as precondition and *ListOfAddresses* as postcondition. A goal for solving the problem is specified as follows:

```
ListOfDocumentNumbers.data->ListOfAddresses.data
```

An additional precondition – a subtask for taking document numbers out of the *ListOfDocumentNumbers* and finding the corresponding addresses for notaries, forms the body for the *Cycle* component. The subtask is specified by connecting data entities of interest to the ports at the bottom of the *Cycle* component. Note that the user does not specify exact services that have to be run repeatedly by the *Cycle*, instead, the user specifies the subtask by its pre- and postconditions. Subtask connections are translated to the logical representation.

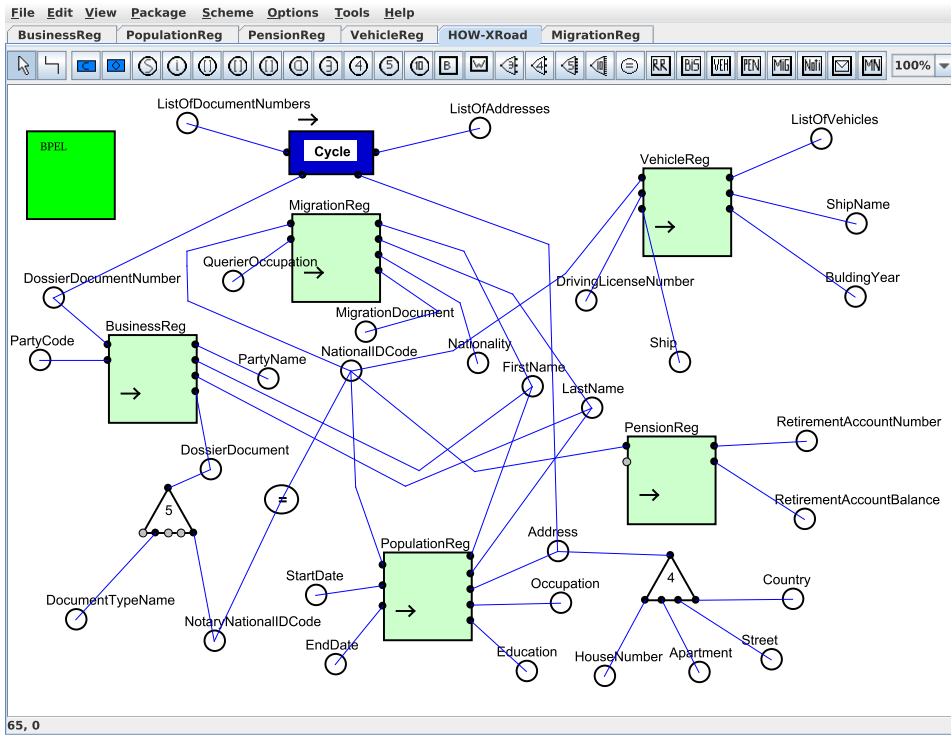


Figure 6.6. Hierarchical X-Road model with higher-order component *Cycle*

As a result the following three services are cycled trough:

arireg:toimiku_dokument (**BusinessReg**)

rr:RR405IsikNimi (**PopulationReg**)

rr:RR57 (**PopulationReg**)

Cycle is translated into BPEL *forEach* construct. It is possible to extend the composition package with other looping constructs. Figure 6.7 visualises the resulting web service in Eclipse BPEL editor. Solving the problem with one subtask contains three services that are run once takes about 0.7 seconds.

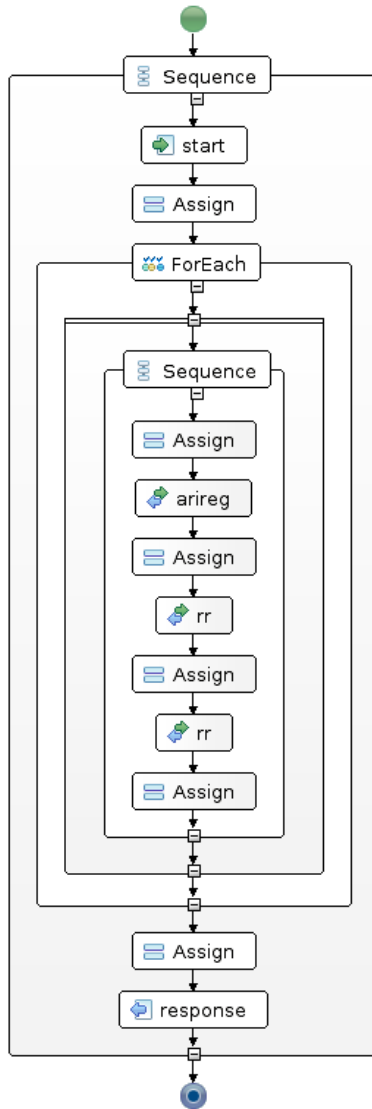


Figure 6.7. Higher-order hierarchical composition visualised in Eclipse BPEL editor

6.3 Conclusion

This chapter showed the usage of hierarchical service models generated from real Estonian e-government web service descriptions. Although hierarchical service models can be very large, they allow to hide some information into submodels and to create views for different users, which enables users to work only with familiar concepts. Experiments in this chapter showed that the proposed method is applicable to service models generated from real-world web service descriptions.

Conclusions and Future Work

A logic-based web service composition method suitable for automating web service composition in knowledge intensive domains with a large number of web services was proposed in this thesis. This method uses service models that are proposed in this thesis as a way to represent available web services, control constructs and data dependency relations. To be able to connect different web services offered by providers that use different ontologies, a hierarchical service model that allows to create views with smaller number of concepts was proposed. A prototype tool, based on the method proposed, was implemented on the CoCoViLa model-based software development platform and tested on a set of web service descriptions of the Estonian e-government information system.

To demonstrate the applicability of the web service composition method, two kinds of experiments were done. First, the method was tested on an existing large flat service model containing about 900 components. Second, a hierarchical model was generated automatically from SAWSDL descriptions and extended with additional data entities and higher-order components.

The existing large flat service model was performing well, but it was hard to use and did not contain all the information necessary to generate a complete output in business process languages, such as, BPEL or OWL-S. Models that were automatically generated from SAWSDL descriptions, included more information and enabled to divide web services into submodels that were used as components on the hierarchical model. Automated model generation also simplifies coping with service description changes. Hierarchical models were easier to use and allowed to create different views for different users without losing in performance. Because descriptions of real world web services were used, some problems that might occur when automating the service model generation were identified as part of this work.

The web service composition tool showed the feasibility of the approach even in case of large models, however, currently the generation of models is not fully automated. For example, complex data types containing other complex data types and complex data types containing arrays are added manually at present. Some difficulties in automating service model generation from SAWSDL descriptions were identified in the thesis.

In addition, currently higher-order components are added manually to the service model. In order to automate generation of service models with higher-order components, a web service description language should be able to represent higher-order workflows. The existing languages, for example, SAWSDL, BPEL, OWL-S, are not the best solution for describing service models with higher-order

components. Hence, they are also not suitable for complete automation of service model generation.

Therefore, future work includes analysis on how to automatically generate service models that include complex data types. As part of the future work it might be necessary to change how WSDL descriptions are usually generated, in order to make them clearer and to reduce redundancy. Also the problem of generating information about higher-order workflows into the service models should be solved, either by creating a new service model specification language or by extending, for example, SAWSDL with the possibility to describe higher-order workflows.

There are many possibilities to extend the service model in the future. For example, the service model could be extended with quality of service information. This would enable to use of service models for commercial services, where the same functionality might be offered by more than one provider. The problem of security and access rights was also out of the scope of this thesis, although, this is a very important topic in the X-Road and other similar domains with very tight access restrictions. Therefore, extending the service model with additional information about the confidentiality and access rights would also be an interesting question for future research.

Bibliography

- [1] Amazon Web Services. <http://soap.amazon.com/schemas2/AmazonWebServices.wsdl>. [25 April 2011].
- [2] Vikas Agarwal, Girish Chafle, Koustuv Dasgupta, Neeran Karnik, Arun Kumar, Sumit Mittal, and Biplav Srivastava. SynthY: a System for End to End Composition of Web Services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(4):311–339, 2005.
- [3] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint Driven Web Service Composition in METEOR-S. In *IEEE International Conference on Service Computing*, pages 23–30, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
- [4] CoCoVila. <http://www.cs.ioc.ee/cocovila/>. [20 April 2011].
- [5] CoCoViLa documentation (Specification Language). http://www.cs.ioc.ee/cocovila/manual/03_spec_lang.pdf. [20 April 2011].
- [6] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and Dynamic Service Composition in eFlow. In *Proceedings of 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000)*, volume 1789 of Lecture Notes in Computer Science, pages 13–31. Springer-Verlag, 2000.
- [7] Marco Crasso, Juan Manuel Rodriguez, Alejandro Zunino, and Marcelo Campo. Revising WSDL Documents: Why and How. *Internet Computing, IEEE*, 14(5):48–56, 2010.
- [8] Marin Dimitrov, Alex Simov, Vassil Momtchev, and Mihail Konstantinov. WSMO Studio – A Semantic Web Services Modelling Environment for WSMO. In *ESWC '07: Proceedings of the 4th European conference on The Semantic Web*, pages 749–758, Berlin, Heidelberg, 2007. Springer-Verlag.
- [9] John Domingue, Liliana Cabral, Stefania Galizia, Vlad Tanasescu, Alessio Gugliotta, Barry Norton, and Carlos Pedrinaci. IRS-III: A Broker-Based Approach to Semantic Web Services. *Journal of Web Semantics*, 6(2):109–132, 2008.
- [10] Easy WSDL Toolbox. <http://easywsdl.ow2.org/>. [14 March 2011].
- [11] Eclipse BPEL Project. <http://www.eclipse.org/bpel/>. [29 March 2011].

- [12] Estonian Information System. <http://ria.ee/public/publikatsioonid/X-road.pdf>. [3 March 2011].
- [13] Daniel Elenius, Grit Denker, David Martin, Fred Gilham, John Khouri, Shahin Sadaati, and Rukman Senanayake. The OWL-S Editor – a Development Tool for Semantic Web Services. In *the Second European Semantic Web Conference*, 2005.
- [14] Khalid Elgazzar, Ahmed E. Hassan, and Patrick Martin. Clustering WSDL Documents to Bootstrap the Discovery of Web Services. *IEEE International Conference on Web Services*, pages 147–154, 2010.
- [15] Ian Foster, Yong Zhao, Ioan Raicu, and Lu Shiyong. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, nov. 2008.
- [16] Pavel Grigorenko, Ando Saabas, and Enn Tyugu. Visual Tool for Generative Programming. *ACM SIGSOFT Software Engineering Notes*, 30(5):249–252, 2005.
- [17] Hele-Mai Haav, Tanel Tammet, Vello Kadarpiik, Kristiina Kindel, and Marko Kääramees. A Semantic-Based Web Service Composition Framework. In Gabor Magyar, Gabor Knapp, Wita Wojtkowski, W. Gregory Wojtkowski, and Jože Zupančič, editors, *Advances in Information Systems Development*, pages 379–391. Springer US, 2008.
- [18] Armin Haller, Emilia Cimpian, Adrian Mocan, Eyal Oren, and Christoph Bussler. WSMX-A Semantic Service-Oriented Architecture. In *IEEE International Conference on Web Services (ICWS 05)*, pages 321–328, Washington, DC, USA, 2005. IEEE Computer Society.
- [19] Wei Jiang, Charles Zhang, Zhenqiu Huang, Mingwen Chen, Songlin Hu, and Zhiyong Liu. QSynth: A Tool for QoS-aware Automatic Service Composition. In *2010 IEEE International Conference on Web Services (ICWS)*, pages 42–49, july 2010.
- [20] Ahto Kalja, Aleksander Reitsakas, and Niilo Saard. eGovernment in Estonia: Best Practices. In *Technology Management : A Unifying Discipline for Melting the Boundaries*, pages 500–506. IEEE, 2005.
- [21] Ahto Kalja, Tarmo Robal, and Uuno Vallner. Towards information society: Estonian case study. In *Management of Engineering Technology, 2009. PICMET 2009. Portland International Conference on*, pages 3218–3225, aug. 2009.

- [22] Mick Kerrigan and Adrian Mocan. The Web Service Modeling Toolkit. In *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 812–816. Springer Berlin / Heidelberg, 2008.
- [23] Srividya Kona, Ajay Bansal, M. Brian Blake, and Gopal Gupta. Generalized Semantics-Based Service Composition. In *IEEE International Conference on Web Services*, pages 219–227, Washington, DC, USA, 2008. IEEE Computer Society.
- [24] Frank Alexander Kraemer, Haldor Samset, and Rolv Braek. An Automated Method for Web Service Orchestration Based on Reusable Building Blocks. In *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*, pages 262–270, Washington, DC, USA, 2009. IEEE Computer Society.
- [25] Peep K ungas and Marlon Dumas. Cost-Effective Semantic Annotation of XML Schemas and Web Service Interfaces. pages 372–379, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [26] Peep K ungas and Mihhail Matskin. Web Services Analysis: Making Use of Web Service Composition and Annotation. In Riichiro Mizoguchi, Zhongzhi Shi, and Fausto Giunchiglia, editors, *The Semantic Web – ASWC 2006*, volume 4185 of *Lecture Notes in Computer Science*, pages 501–515. Springer Berlin / Heidelberg.
- [27] Peep K ungas and Mihhail Matskin. From web services annotation and composition to web services domain analysis. *Int. J. Metadata Semant. Ontologies*, 2:157–178, March 2007.
- [28] Sven L ammermann. *Runtime Service Composition via Logic-Based Program Synthesis*. PhD thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, 2002.
- [29] Xuanzhe Liu, Gang Huang, and Hong Mei. A User-Oriented Approach to Automated Service Composition. In *IEEE International Conference on Web Services, 2008. ICWS '08.*, pages 773–776, sept. 2008.
- [30] Riina Maigre. Survey of the Tools for Automating Service Composition. In *2010 IEEE International Conference on Web Services (ICWS 2010)*, pages 628–629. IEEE Computer Society, 2010.
- [31] Riina Maigre, Pavel Grigorenko, Peep K ungas, and Enn Tyugu. Stratified Composition of Web Services. In *Proceeding of the 2008 conference on Knowledge-Based Software Engineering: Proceedings of the Eighth Joint Conference on Knowledge-Based Software Engineering*, pages 49–58, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

- [32] Riina Maigre, Peep Kúngas, Mihhail Matskin, and Enn Tyugu. Handling Large Web Services Models in a Federated Governmental Information System. In *ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, pages 626–631, Washington, DC, USA, 2008. IEEE Computer Society.
- [33] Riina Maigre, Peep Kúngas, Mihhail Matskin, and Enn Tyugu. Dynamic Service Synthesis on Large Service Models of a Federated Governmental Information System. *International Journal On Advances in Intelligent Systems*, 2(1):182–191, 2009. http://www.iariajournals.org/intelligent_systems/.
- [34] Riina Maigre and Enn Tyugu. Composition of Services on Hierarchical Service Models. In *The 21st European - Japanese Conference on Information Modelling and Knowledge Bases (EJC 2011)*, 2011. [in print].
- [35] Shalil Majithia, Matthew S. Shields, Ian J. Taylor, and Ian Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 514–524. IEEE Computer Society, 2004.
- [36] Alberto Martínez, Marta Patiño Martínez, Ricardo Jiménez-Peris, and Francisco Pérez-Sorrosal. ZenFlow: a Visual Web Service Composition Tool for BPEL4WS. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 181–188, 2005.
- [37] Mihhail Matskin, Riina Maigre, and Enn Tyugu. Compositional Logical Semantics for Business Process Languages. In *Proceedings of Second International Conference on Internet and Web Applications and Services (ICIW 2007)*. IEEE Computer Society, 2007.
- [38] Mihhail Matskin and Enn Tyugu. Strategies of Structural Synthesis of Programs and its Extensions. *Computing and Informatics*, 20:1–25, 2001.
- [39] Grigori Mints. *A Short Introduction to Intuitionistic Logic*. University Series in Mathematics. Springer, 2001.
- [40] Grigori Mints and Enn Tyugu. Justifications of the Structural Synthesis of Programs. *Science of Computer Programming*, 2(3):215–240, 1982.
- [41] Richi Nayak and Bryan Lee. Web Service Discovery with Additional Semantics and Clustering. In *WI '07: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pages 555–558, Washington, DC, USA, 2007. IEEE Computer Society.
- [42] OWL-S: Semantic Markup for Web Services. <http://www.w3.org/Submission/OWL-S/>. [13 March 2011].

- [43] Thomas Oinn, Mark Greenwood, Matthew Addis, Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Christopher Wroe. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, August 2006.
- [44] Cesare Pautasso and Gustavo Alonso. JOpera: a Toolkit for Efficient Visual Composition of Web Services. *International Journal of Electronic Commerce*, 9(2):107–141, 2004.
- [45] J. Peer. Web Service Composition as AI Planning – a Survey. *University of St. Gallen, Switzerland*, 2005.
- [46] Charles Petrie and Christoph Bussler. The Myth of Open Web Services: The Rise of the Service Parks. *IEEE Internet Computing*, 12(3):96–95, 2008.
- [47] Marco Pistore, Paolo Traverso, Piegiorgio Bertoli, and Annapaola Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *Proceedings of 2005 IEEE International Conference on Web Services (ICWS 2005)*, pages 293–301, 2005.
- [48] Shankar R. Ponnekanti and Armando Fox. Sword: A Developer Toolkit for Web Service Composition. In *WWW '02: Proceedings of the 11th International World Wide Web Conference*, 2002.
- [49] The administration system for state information system (RIHA). <https://riha.eesti.ee>. [2 March 2011].
- [50] Jinghai Rao, Peep Küngas, and Mihhail Matskin. Composition of Semantic Web Services Using Linear Logic Theorem Proving. *Information Systems, Special Issue on the Semantic Web and Web Services*, 31(4–5):340–360, 2006.
- [51] Jinghai Rao and Xiaomeng Su. A Survey of Automated Web Service Composition Methods. In Jorge Cardoso and Amit P. Sheth, editors, *SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2004.
- [52] John W. Rittinghouse and James F. Ransome. *Cloud Computing: Implementation, Management, and Security*. Auerbach Publications, 2010. Books24x7. http://common.books24x7.com/book/id_32102/book.asp. [24 April 2011].
- [53] Semantic Annotations for WSDL and XML Schema. <http://www.w3.org/TR/sawSDL/>. [13 March 2011].

- [54] James Scicluna, Charlie Abela, and Matthew Montebello. Visual Modelling of OWL-S Services. In *IADIS International Conference WWW/Internet*, 2004.
- [55] Evren Sirin, James Hendler, and Bijan Parsia. Semi-Automatic Composition of Web Services Using Semantic Descriptions. In *Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, pages 17–24, 2002.
- [56] Michele Trainotti, Marco Pistore, Gaetano Calabrese, Gabriele Zacco, Gigi Lucchese, Fabio Barbon, Piergiorgio Bertoli, and Paolo Traverso. AS-TRO: Supporting Composition and Execution of Web Services. In *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 495–501. Springer Berlin / Heidelberg, 2005.
- [57] Enn Tyugu. The Structural Synthesis of Programs. In *Algorithms in Modern Mathematics and Computer Science*, pages 82–99, London, UK, 1981. Springer-Verlag.
- [58] Enn Tyugu. *Algorithms and Architectures of Artificial Intelligence*. IOS Press, 2007.
- [59] Enn Tyugu. Grigori Mints and computer science. In Solomon Feferman and Wilfried Sieg, editors, *Proofs, Categories and Computations: Essays in Honor of Grigori Mints*, pages 267–277. London: College Publications, 2010.
- [60] Tomas Vitvar, Mick Kerrigan, Arnold van Overeem, Vassilios Peristeras, and Konstantinos Tarabanis. Infrastructure for the Semantic Pan-European E-government Services. In *Proceedings of the 2006 AAAI Spring Symposium on The Semantic Web meets eGovernment (SWEG)*, 3 2006.
- [61] Web Service Modeling Language. <http://www.wsmo.org/wsml/>. [16 April 2011].
- [62] Web Service Modeling Ontology. <http://www.wsmo.org/>. [16 April 2011].
- [63] Web Service Semantics – WSDL-S. <http://www.w3.org/Submission/WSDL-S/>. [25 April 2011].
- [64] Web Services Business Process Execution Language Version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>. [13 March 2011].
- [65] Web Services Choreography Description Language Version 1.0. <http://www.w3.org/TR/ws-cdl-10/>. [5 April 2011].

- [66] Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsd1>. [15 March 2011].
- [67] Bruno Wassermann, Wolfgang Emmerich, Ben Butchart, Nick Cameron, Liang Chen, and Jignesh Patel. Sedna: a BPEL-Based Environment for Visual Scientific Workflow Modelling. In I.J. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for eScience – Scientific Workflows for Grids*. Springer Verlag, 2007.
- [68] Requirements on information systems and adapter servers. http://ftp.ria.ee/pub/x-tee/doc/nouded_infosysteemidele_en.pdf. [Specification date: 10 June 2005].
- [69] XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xpath/>. [27 March 2011].
- [70] yEd Graph Editor. http://www.yworks.com/en/products_yed_about.html. [2 March 2011].
- [71] Interoperability Solutions for European Public Administrations. <http://ec.europa.eu/isa/>. [10 April 2011].
- [72] Oracle BPEL Process Manager. <http://www.oracle.com/technology/bpel>. [28 April 2010].
- [73] Information Technology in Public Administration of Estonia. Yearbook 2006, Estonian Ministry of Economic Affairs and Communication, 2007.
- [74] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

List of Publications

- Riina Maigre, Enn Tyugu (2011). Composition of Services on Hierarchical Service Models. In: EJC 2011: 21st European-Japanese Conference on Information Modelling and Knowledge Bases. [in print]
- Riina Maigre (2010). Survey of the Tools for Automating Service Composition. In: ICWS 2010: 2010 IEEE Eighth International Conference on Web Services: Miami, Florida, 5–10 July 2010: IEEE Computer Society, 2010, 628–629.
- Riina Maigre, Peep Kúngas, Mihhail Matskin, Enn Tyugu (2009). Dynamic Service Synthesis on a Large Service Models of a Federated Governmental Information System. *International Journal on Advances in Intelligent Systems*, 2(1), 181–191.
- Riina Maigre, Pavel Grigorenko, Peep Kúngas, Enn Tyugu (2008). Stratified Composition of Web Services. In: *Knowledge-based software engineering: Proceedings of the Eighth Joint Conference on Knowledge-Based Software Engineering: (Eds.) Virvou, Maria; Nakamura, Taichi*. Amsterdam: IOS Press, 2008, (Frontiers in Artificial Intelligence and Applications; 180), 49–58.
- Riina Maigre, Peep Kúngas, Mihhail Matskin, Enn Tyugu (2008). Handling Large Web services models in a Federated Governmental Information System. In: *The Third International Conference on Internet and Web Applications and Services, ICIW 2008: 8–13 June 2008, Athens, Greece, proceedings: (Eds.) Mellouk, A.; Bi, J.; Ortiz, G. et al*. Los Alamitos: IEEE Computer Society, 2008, 626–631.
- Mihhail Matskin, Riina Maigre, Enn Tyugu (2007). Compositional Logical Semantics for Business Process Languages. In: *Second International Conference on Internet and Web applications and services ICIW 2007, 13–19 May 2007, Morne, Mauritius: Los Alamitos, CA: IEEE Computer Society, 2007, 6 p.*

Veebiteenuste kompositsioon suurtel teenustemudelitel

Annotatsioon

Käesolev väitekiri uurib uute kompleksveebiteenuste automaatset konstrueerimist reaalses teadmistemahukas valdkonnas suure hulga eri ontoloogiaid kasutavate baasveebiteenuste korral. Teenuste kompositsiooni eesmärk on konstrueerida olemasolevatest veebiteenustest uusi veebiteenuseid, et lahendada ülesandeid, mida esialgsed veebiteenused ei võimaldanud. Kompositsiooni muudab sellises valdkonnas keeruliseks ühest küljest väga suur baasveebiteenuste hulk, millest komponendid tuleb valida. Teisest küljest võivad veebiteenused, mida on vaja kompositsiooni lisada, kasutada erinevaid ontoloogiaid, mistõttu automatiseerimine on raske. Lisaks võib väga spetsiifilise ja keerulise valdkonna korral valdkonna spetsialist, kes ei pruugi olla programmeerija, disainida parema kompleksteenuse, kui seda teeks programmeerijast veebiteenuste arendaja või automaatne kompositsiooni tööriist. Sellisel juhul on vaja kergesti kasutatavaid tööriistu, mille rakendamiseks saab valdkonna spetsialist hakkama. Veebiteenuste automaatne kompositsioon on väga oluline ka Eesti riigi jaoks, sest riigi infosüsteem koosneb juba praegu rohkem kui kahest tuhandest veebiteenusest, kuid nende automaatseks kompositsiooniks puuduvad veel tööriistad.

Käesolev doktoritöö defineerib teenustemudeli mõiste ning kasutab teenustemudelite suure hulga olemasolevate veebiteenuste, ontoloogiamõistete ning juhtstruktuuride kirjeldamiseks. Teenustemudelil on visuaalne esitus, mida saab kasutada valdkonna ekspert, ning loogikapõhine esitus, mida kasutab planeerija automaatseks veebiteenuste sünteesiks. Teenustemudeli formaalseks kirjeldamiseks kasutatakse intuitsionistlikku lauseloogikat. Juhtstruktuure sisaldavatel suurtel mudelitel uute veebiteenuste automaatseks sünteesiks kasutatakse struktuurset programmide sünteesi, mis põhineb intuitsionistlikul lauseloogikal.

Sünteesimeetodi praktilist rakendatavust ja skaleeruvust on katsetatud Eesti riigi infosüsteemi veebiteenuste põhjal valminud suurel teenustemudelil. Kuigi süntees sellel suurel mudelil toimub kiiresti, on vajalik, et veebiteenused kasutaks kõiki teenusepakkujaid hõlmavat ühist ontoloogiat.

Suure teenustemudeli keerukuse vähendamiseks ning erinevaid ontoloogiaid kasutavate teenustemudelite ühendamiseks on välja pakutud teenustemudeli hierarhiline versioon. Hierarhiline mudel võimaldab jagada veebiteenused alamudelitesse ning kasutada alamudelite hierarhilise mudeli komponentidena. See võimaldab kasutada erinevaid ontoloogiaid eri alamudelitel jaoks ning tuua hierarhilise mudeli ülemisel tasemel välja vaid kasutajale olulised mõisted.

Composition of Web Services on Large Service Models

Abstract

This thesis addresses the task of building new web services automatically in a real world knowledge intensive domain with a large number of available web services using different ontologies. The aim of web service composition is to construct new, more complex web services from existing ones, in order to solve tasks that existing web services could not solve on their own. The manual composition process gets very complicated when the set of web services to choose from is large. Automated composition, on the other hand, is hard to archive because web services that have to be composed might use different ontologies. In addition, in the case of a very specialised and knowledge intensive domain, a domain expert, who might not be a programmer, may be able to design better compound web services than a web service developer or an automated composition tool. In this case, visual composition tools are needed that are simple enough, so that domain experts are able to use them. Automated web service composition is also important for Estonia, because the Estonian e-government information system contains already more than 2000 web services, but currently there are no tools that allow to compose these web services automatically.

A concept of service model is defined and the usage of service models is proposed in this thesis as a way to describe a large number of available web services, concepts from ontology and control constructs. The service model has a visual representation that can be used by a domain expert, and a logical representation that is used by a planner to automate web service composition. Intuitionistic propositional logic is used for formal description of service models, and structural synthesis of programs based on this logic is applied for automating the web service composition on large service models that include control constructs.

Experiments have been conducted on a large service model built from the Estonian e-government web service descriptions in order to show the practical applicability and scalability of the synthesis method. Although planning on this large model had a good performance, it required the usage of a single large ontology for all web service providers.

To reduce the complexity of this large service model, and to be able to connect web services using different ontologies, a hierarchical service model was developed. The hierarchical model enables the splitting of a set of web services into submodels and the use of submodels as components of a hierarchical model. This allows to use different ontologies for different submodels and to bring out relevant concepts on the upper level model.

ELULOOKIRJELDUS

1. Isikuandmed

Ees- ja perekonnanimi: Riina Maigre
Sünniaeg ja -koht: 24. oktoober 1981, Tallinn, Eesti
Kodakondsus: Eesti

2. Kontaktandmed

Address: Akadeemia tee 21, 12618, Tallinn
Telefon: 620 4224
E-posti aadress: riina@ioc.ee

3. Hariduskäik

Õppeasutus (nimetus lõpetamise ajal)	Lõpetamise aeg	Haridus (eriala/kraad)
Tallinna Tehnikaülikool	2007	informaatika/tehnikateaduste magister
Tallinna Tehnikaülikool	2005	Võrgutarkvara ja intelligentsed süsteemid/diplom

4. Keelteoskus (alg-, kesk- või kõrgtase)

Keel	Tase
eesti keel	emakeel
inglise keel	kõrgtase
vene keel	algtase

5. Täiendusõpe

Õppimise aeg	Täiendusõppe läbiviija nimetus
1.–6. juuni 2009	9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services (SFM-09:WS)
26.–30. august 2007	6th Estonian Summer School in Computer and Systems Science (ESSCaSS'07)
6.–10. august 2006	5th Estonian Summer School in Computer and Systems Science (ESSCaSS'06)

6. Teenistuskäik

Töötamise aeg	Tööandja nimetus	Ametikoht
2009–...	Tallinna Tehnikaülikooli Küberneetika Instituut	teadur
2005–...	Tallinna Tehnikaülikooli Küberneetika Instituut	arvutisüsteemi administraator

7. Teadustegevus

Riina Maigre, Enn Tyugu (2011). Composition of services on hierarchical service models. In: EJC 2011: 21st European-Japanese Conference on Information Modelling and Knowledge Bases. [ilmumas]

Vahur Kotkas, Andres Ojamaa, Pavel Grigorenko, Riina Maigre, Mait Harf, Enn Tyugu (2011). CoCoViLa as a multifunctional simulation platform. In: SIMU-TOOLS 2011: 4th International ICST Conference on Simulation Tools and Techniques: 21–25 March 2011, Barcelona, Spain: Brussels: ICST, 2011, 1–8.

Riina Maigre (2010). Survey of the tools for automating service composition. In: ICWS 2010: 2010 IEEE Eighth International Conference on Web Services: Miami, Florida, 5–10 July 2010: IEEE Computer Society, 2010, 628–629.

Riina Maigre, Peep Küngas, Mihhail Matskin, Enn Tyugu (2009). Dynamic service synthesis on a large service models of a federated governmental information system. *International Journal on Advances in Intelligent Systems*, 2(1), 181–191.

Riina Maigre, Pavel Grigorenko, Peep Küngas, Enn Tyugu (2008). Stratified composition of web services. In: *Knowledge-based software engineering: Proceedings of the Eighth Joint Conference on Knowledge-Based Software Engineering: (Toim.) Virvou, Maria; Nakamura, Taichi*. Amsterdam: IOS Press, 2008, (Frontiers in Artificial Intelligence and Applications; 180), 49–58.

Riina Maigre, Peep Küngas, Mihhail Matskin, Enn Tyugu (2008). Handling large web services models in a federated governmental information system. In: *The Third International Conference on Internet and Web Applications and Services, ICIW 2008: 8–13 June 2008, Athens, Greece, proceedings: (Toim.) Mellouk, A.; Bi, J.; Ortiz, G. et al*. Los Alamitos: IEEE Computer Society, 2008, 626–631.

Mihhail Matskin, Riina Maigre, Enn Tyugu (2007). Compositional logical semantics for business process languages. In: Second International Conference on Internet and Web applications and services ICIW 2007, 13–19 May 2007, Morne, Mauritius: Los Alamitos, CA: IEEE Computer Society, 2007, 6 p.

Adam Eppendahl, Riina Maigre (2005). Mobile camera parameter recovery in an unknown environment without point features. In: Proceedings 2005 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA 2005: 27–30 June 2005, Espoo, Finland: Piscataway, N.J.: IEEE, 2005, 279–283.

8. Kaitstud lõputööd

Visuaalse kasutajaliidesega veebiteenuste kompositsioonitarkvara, Tallinna Tehnikaülikool, Küberneetika Instituut, 2007, juhendaja Enn Tõugu.

Valgusväljamudelitest liikuva kaamera ja sõiduki parameetrite tuletamine. Tallinna Tehnikaülikool, Arvutiteaduse Instituut, 2005, juhendajad Adam Eppendahl, Juhan Ernits.

9. Teadustöö põhisuunad

Veebiteenused, veebiteenuste automaatne kompositsioon, veebiteenuste semantika.

CURRICULUM VITAE

1. Personal data

Name: Riina Maigre
Date and place of birth: 24 October 1981, Tallinn, Estonia

2. Contact information

Address: Akadeemia tee 21, 12618, Tallinn, Estonia
Phone: +372 620 4224
E-mail: riina@ioc.ee

3. Education

Educational institution	Graduation year	Education (field of study/degree)
Tallinn University of Technology	2007	Informatics/Master of Science in Engineering
Tallinn University of Technology	2005	Network software and intelligent systems/diploma

4. Language competence/skills (fluent, average, basic skills)

Language	Level
Estonian	fluent
English	fluent
Russian	basic skills

5. Special Courses

Period	Educational or other information
1–6 June 2009	9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services (SFM-09:WS)
26–30 August 2007	6th Estonian Summer School in Computer and Systems Science (ESSCaSS'07)
6–10 August 2006	5th Estonian Summer School in Computer and Systems Science (ESSCaSS'06)

6. Professional Employment

Period	Organisation	Position
2009–...	Institute of Cybernetics at Tallinn University of Technology	researcher
2005–...	Institute of Cybernetics at Tallinn University of Technology	system administrator

7. Scientific work

Riina Maigre, Enn Tyugu (2011). Composition of services on hierarchical service models. In: EJC 2011: 21st European-Japanese Conference on Information Modelling and Knowledge Bases. [in print]

Vahur Kotkas, Andres Ojamaa, Pavel Grigorenko, Riina Maigre, Mait Harf, Enn Tyugu (2011). CoCoViLa as a multifunctional simulation platform. In: SIMU-TOOLS 2011: 4th International ICST Conference on Simulation Tools and Techniques: 21–25 March 2011, Barcelona, Spain: Brussels: ICST, 2011, 1–8.

Riina Maigre (2010). Survey of the tools for automating service composition. In: ICWS 2010: 2010 IEEE Eighth International Conference on Web Services: Miami, Florida, 5–10 July 2010: IEEE Computer Society, 2010, 628–629.

Riina Maigre, Peep KÜngas, Mihhail Matskin, Enn Tyugu (2009). Dynamic service synthesis on a large service models of a federated governmental information system. *International Journal on Advances in Intelligent Systems*, 2, 181–191.

Riina Maigre, Pavel Grigorenko, Peep KÜngas, Enn Tyugu (2008). Stratified composition of web services. In: *Knowledge-based software engineering: Proceedings of the Eighth Joint Conference on Knowledge-Based Software Engineering: (Eds.)Virvou, Maria; Nakamura, Taichi*. Amsterdam: IOS Press, 2008, (Frontiers in Artificial Intelligence and Applications; 180), 49–58.

Riina Maigre, Peep KÜngas, Mihhail Matskin, Enn Tyugu (2008). Handling large web services models in a federated governmental information system. In: *The Third International Conference on Internet and Web Applications and Services, ICIW 2008: 8–13 June 2008, Athens, Greece, proceedings: (Eds.)Mellouk, A.; Bi, J.; Ortiz, G. et al*. Los Alamitos: IEEE Computer Society, 2008, 626-631.

Mihhail Matskin, Riina Maigre, Enn Tyugu (2007). Compositional logical semantics for business process languages. In: Second International Conference on Internet and Web applications and services ICIW 2007, 13–19 May 2007, Morne, Mauritius: Los Alamitos, CA: IEEE Computer Society, 2007, 6 p.

Adam Eppendahl, Riina Maigre (2005). Mobile camera parameter recovery in an unknown environment without point features. In: Proceedings 2005 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA 2005: 27–30 June 2005, Espoo, Finland: Piscataway, N.J.: IEEE, 2005, 279–283.

8. Defended theses

Web service composition software with visual user interface, Institute of Cybernetics at Tallinn University of Technology, 2007. Supervisor Enn Tõugu.

Using light field models to recover mobile camera and vehicle parameters, Tallinn University of Technology, Department of Computer Science, 2005. Supervisors Adam Eppendahl, Juhan Ernits.

9. Main areas of scientific work/Current research topics

Web services, automated composition of web services, semantics of web services.

**DISSERTATIONS DEFENDED AT
TALLINN UNIVERSITY OF TECHNOLOGY ON
*INFORMATICS AND SYSTEM ENGINEERING***

1. **Lea Elmik**. Informational modelling of a communication office. 1992.
2. **Kalle Tammemäe**. Control intensive digital system synthesis. 1997.
3. **Eerik Lossmann**. Complex signal classification algorithms, based on the third-order statistical models. 1999.
4. **Kaido Kikkas**. Using the Internet in rehabilitation of people with mobility impairments – case studies and views from Estonia. 1999.
5. **Nazmun Nahar**. Global electronic commerce process: business-to-business. 1999.
6. **Jevgeni Riipulk**. Microwave radiometry for medical applications. 2000.
7. **Alar Kuusik**. Compact smart home systems: design and verification of cost effective hardware solutions. 2001.
8. **Jaan Raik**. Hierarchical test generation for digital circuits represented by decision diagrams. 2001.
9. **Andri Riid**. Transparent fuzzy systems: model and control. 2002.
10. **Marina Brik**. Investigation and development of test generation methods for control part of digital systems. 2002.
11. **Raul Land**. Synchronous approximation and processing of sampled data signals. 2002.
12. **Ants Ronk**. An extended block-adaptive Fourier analyser for analysis and reproduction of periodic components of band-limited discrete-time signals. 2002.
13. **Toivo Paavle**. System level modeling of the phase locked loops: behavioral analysis and parameterization. 2003.
14. **Irina Astrova**. On integration of object-oriented applications with relational databases. 2003.
15. **Kuldar Taveter**. A multi-perspective methodology for agent-oriented business modelling and simulation. 2004.
16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.
17. **Artur Jutman**. Selected issues of modeling, verification and testing of digital systems. 2004.
18. **Ander Tenno**. Simulation and estimation of electro-chemical processes in maintenance-free batteries with fixed electrolyte. 2004.
19. **Oleg Korolkov**. Formation of diffusion welded Al contacts to semiconductor silicon. 2004.
20. **Risto Vaarandi**. Tools and techniques for event log analysis. 2005.
21. **Marko Koort**. Transmitter power control in wireless communication systems. 2005.

22. **Raul Savimaa.** Modelling emergent behaviour of organizations. Time-aware, UML and agent based approach. 2005.
23. **Raido Kurel.** Investigation of electrical characteristics of SiC based complementary JBS structures. 2005.
24. **Rainer Taniloo.** Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.
25. **Pauli Lallo.** Adaptive secure data transmission method for OSI level I. 2005.
26. **Deniss Kumlander.** Some practical algorithms to solve the maximum clique problem. 2005.
27. **Tarmo Veskioja.** Stable marriage problem and college admission. 2005.
28. **Elena Fomina.** Low power finite state machine synthesis. 2005.
29. **Eero Ivask.** Digital test in WEB-based environment 2006.
30. **Виктор Войтович.** Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.
31. **Tanel Alumäe.** Methods for Estonian large vocabulary speech recognition. 2006.
32. **Erki Eessaar.** Relational and object-relational database management systems as platforms for managing softwareengineering artefacts. 2006.
33. **Rauno Gordon.** Modelling of cardiac dynamics and intracardiac bio-impedance. 2007.
34. **Madis Listak.** A task-oriented design of a biologically inspired underwater robot. 2007.
35. **Elmet Orasson.** Hybrid built-in self-test. Methods and tools for analysis and optimization of BIST. 2007.
36. **Eduard Petlenkov.** Neural networks based identification and control of nonlinear systems: ANARX model based approach. 2007.
37. **Toomas Kirt.** Concept formation in exploratory data analysis: case studies of linguistic and banking data. 2007.
38. **Juhan-Peep Ernits.** Two state space reduction techniques for explicit state model checking. 2007.
39. **Innar Liiv.** Pattern discovery using seriation and matrix reordering: A unified view, extensions and an application to inventory management. 2008.
40. **Andrei Pokatilov.** Development of national standard for voltage unit based on solid-state references. 2008.
41. **Karin Lindroos.** Mapping social structures by formal non-linear information processing methods: case studies of Estonian islands environments. 2008.
42. **Maksim Jenihhin.** Simulation-based hardware verification with high-level decision diagrams. 2008.

43. **Ando Saabas.** Logics for low-level code and proof-preserving program transformations. 2008.
44. **Ilja Tšahhirov.** Security protocols analysis in the computational model – dependency flow graphs-based approach. 2008.
45. **Toomas Ruuben.** Wideband digital beamforming in sonar systems. 2009.
46. **Sergei Devadze.** Fault Simulation of Digital Systems. 2009.
47. **Andrei Krivošei.** Model based method for adaptive decomposition of the thoracic bio-impedance variations into cardiac and respiratory components.
48. **Vineeth Govind.** DFT-based external test and diagnosis of mesh-like networks on chips. 2009.
49. **Andres Kull.** Model-based testing of reactive systems. 2009.
50. **Ants Torim.** Formal concepts in the theory of monotone systems. 2009.
51. **Erika Matsak.** Discovering logical constructs from Estonian children language. 2009.
52. **Paul Annus.** Multichannel bioimpedance spectroscopy: instrumentation methods and design principles. 2009.
53. **Maris Tõnso.** Computer algebra tools for modelling, analysis and synthesis for nonlinear control systems. 2010.
54. **Aivo Jürgenson.** Efficient semantics of parallel and serial models of attack trees. 2010.
55. **Erkki Joason.** The tactile feedback device for multi-touch user interfaces. 2010.
56. **Jürgo-Sören Preden.** Enhancing situation – awareness cognition and reasoning of ad-hoc network agents. 2010.
57. **Pavel Grigorenko.** Higher-Order Attribute Semantics of Flat Languages. 2010.
58. **Anna Rannaste.** Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.
59. **Sergei Strik.** Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.
60. **Rain Ottis.** A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.
61. **Natalja Sleptšuk.** Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.
62. **Martin Jaanus.** The Interactive Learning Environment for Mobile Laboratories. 2011.
63. **Argo Kasemaa.** Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.
64. **Kenneth Geers.** Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

