Composition of Services on Hierarchical Service Models

Riina MAIGRE[a,1], Enn TYUGU[a,b]
[a]Institute of Cybernetics at Tallinn University of Technology, Estonia
[b]NATO Cooperative Cyber Defence Center of Excellence

Abstract. In this paper we introduce hierarchical service models suitable for large service parks. We describe a method for handling higher-order workflows that we use for automatic composition of services. We give a description of a specification language suitable for representing hierarchical service models. We introduce two kinds of ontologies: service ontologies and user ontologies, and demonstrate their usage in our experiments on Estonian e-government information system.

Keywords. Hierarchical service models, synthesis on hierarchical service models, e-government services, service parks.

Introduction

The approach to synthesis of services presented in this paper is based on an assumption that services can be collected in service parks [1]. Services of a service park can be developed with good interoperability, and described in a simple (restricted) ontology. The paradigm of service parks can be applied, in particular, to the development of governmental services to citizens, as well as to services in military network centric control and command systems that are under development in many countries at present.

Estonian governmental institutions have an obligation to provide numerous public services over Internet in different areas called eSchool, eHealth, etc. The services have been developed by different authorities (ministries) and are based on domain databases maintained by them. The number of atomic services is large (over 2000 services). New compound services must be developed constantly, as well as existing services have to be improved and updated. In particular, new services with cross-domain access of services are required. We have developed and tested a prototype of automatic composition of new services in our earlier work [4]. It supports the cross-domain service composition as well. It is based on a sound logic and performs technically well. However, the usage of the prototype required a knowledge of a large joint ontology of all services of all ministries that is difficult for users.

We introduce hierarchical models for service composition now. This simplifies the ontologies to be used and will facilitate the development and usage of new services. We give a description of a specification language for service composition, and describe experiments on Estonian e-government information system. This work builds on top of

1 Corresponding Author: Riina Maigre; E-mail: riina@ioc.ee

our previous work on logical semantics for process languages [3] and on the experiments with service synthesis on large federated governmental models [4].

The paper is structured as follows. Ontologies for services and for users are described in Section 1. Service model is introduced in Section 2, followed by the hierarchical service model description in Section 3. Higher-order workflows are discussed in Section 4. Section 5 gives the representation of higher-order workflows and service models in logic. Section 6 introduces the specification language for service composition. Usage of specification language is shown on examples in Section 7. Experiments with the hierarchical service models of Estonian e-government are described in Section 8 and related work is given in Section 9. Conclusion is given in Section 10.

1. Ontologies for Services and for Users

We use ontologies as modules of shared knowledge, developing them with clear understanding of the needs of their users. There are ontologies for different public domains: health, education, transportation, police etc. These ontologies include detailed information about usable data entities, for example, *EntryDate*, *RegistryCode*, *EntryTypeInBusinessRegistry*, as well as logical propositions that control service composition – *EndOfCalculations*. As these names belong to a specific name space of a particular domain (business registry in this case) their meaning is relatively easy to explain and to understand. These ontologies are needed for describing atomic services available for a domain, therefore we call them *service ontologies*. Besides the service ontologies of public domains we use one general ontology that includes only the information for composing services from different domains, and no details about the internal details of any domain. We call it a *user ontology*. This ontology includes only these concepts of domain ontologies that must be visible for a user of services. An example concept of this ontology is *NationalIDCode* for identifying a person that is used in many domains. Conceptually, this ontology includes external views of all service ontologies, and additionally some concepts for controlling the composition of services.

2. 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. It can describe a service park. Services in one service model have to share a common ontology and a background information. Services from different service parks that share a common ontology can be represented in one and the same service model. We represent a service model as a bipartite graph with two sets of nodes R and V. The set R is a set of services and data dependency relations. 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 5. 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 the transformation component.

A data dependency 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'',v2)*. An abstract service is implemented by the synthesizer, it does not have an explicit grounding.

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. If the model is used for composition, composed service is correct with respect to the service model.

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

Our service model approach differs from the most logic-based approaches, as in addition to obvious input and output mapping, we explicitly take into account the domain's background information when creating a service model. Besides that, we allow the user to extend the model with customized higher-order or simple services. That is, workflow language is not limited to usual workflow constructs like sequence, cycle and condition, 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.

3. Hierarchical Service Model

Given a set of semantically annotated web service descriptions, process of developing a hierarchical service model begins with generating service models for each web service description. If semantic annotations are good enough then this part can be fully automated. We have currently automated the generation of service models from semantic annotations for WSDL (SAWSDL)[2]. Models created from semantic annotations can be used as components in the hierarchical service models. Creation of hierarchical models can also be partially automated, but in general we expect the domain expert to work at 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. The relations between semantically annotated service descriptions, service models and a hierarchical views is shown on Figure 1.

In the first layer in Figure 1 we have annotated service descriptions given, in SAWSDL. Service models representing atomic web services and data-dependencies between services are generated from these descriptions. Small rectangles in Layer 2 represent atomic services (WSDL operations) and lines between them represent data

2 SAWSDL – http://www.w3.org/TR/sawsdl/

paths. Service models are saved as components and used in the models in Layer 3. Note that there can be more than one hierarchical model and that, depending on the task's context, different hierarchical models can contain different service models as components. Therefore, we call these hierarchical models hierarchical views.

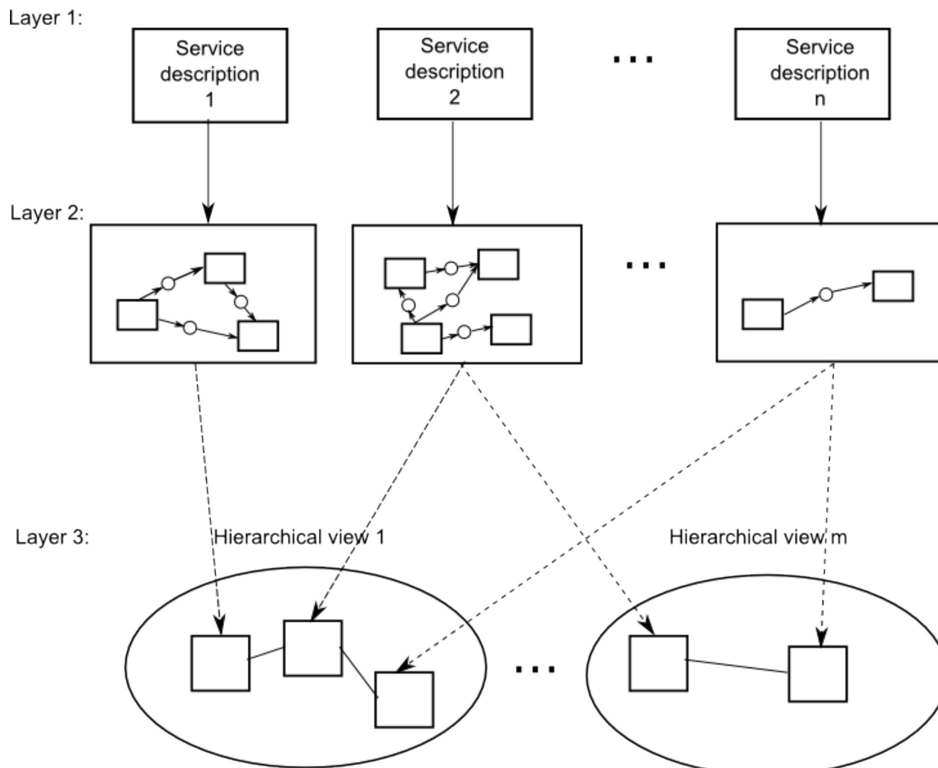

Figure 1: Relations between annotated service descriptions and service models and hierarchical views.

4. Higher-Order Workflows (HOW)

In this and in the following section we present higher-order workflow (HOW) and a logic that enables us to represent service models and to compose services automatically. The approach extends conventional workflow semantics by enabling the specification of higher order service models. We are going to represent the logic on an example and try to avoid purely syntactic formal representation as much as possible. The example in this section is based on the material of our publication [3].

Let us consider an example of a travel planning service. We assume that this is a composite service that supports students in planning a trip around Europe during their summer vacations. We are going to use the existing atomic services *chooseLeg* and *addLeg* for making a choice of a new travel leg and adding it to the itinerary. There also has to be a payment service (*finish*), e.g., giving the credit card number and authorizing the purchase. In process and workflow languages, we have control constructs *sequence*, *cycle* and other that vary from language to language to some

extent, but can all be easily implemented in our case. Because the services *chooseLeg* and *addLeg* must be repeated several times (the number of repetitions is not known beforehand), our composite service should introduce a control over services that forms a cycle. We can represent the workflow very abstractly as a graph shown in Figure 2.

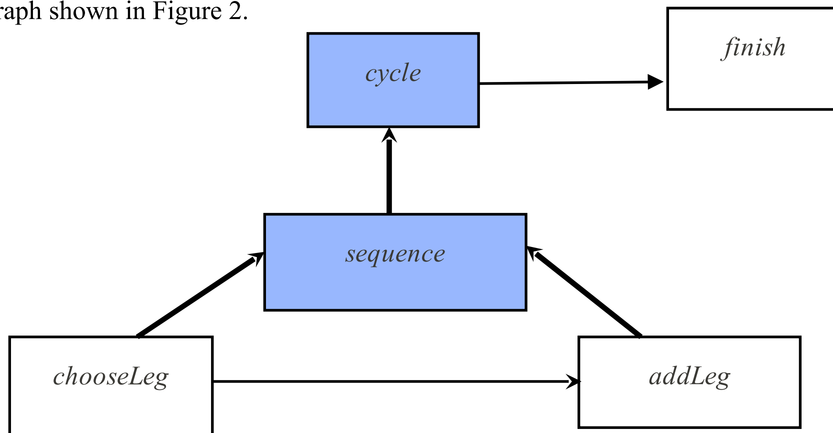

Figure 2. Higher-order workflow

In this case we have two types of services in a composite service workflow: basic, and control or structural. Basic services (white rectangles) are atomic services (for example, travel service for planning one leg of the trip) and control services (blue rectangles) are loops, choice, sequences or other possible control patterns over other services. Thin lines in Figure 2 express the ordering of the invocation of services and thick lines express control that *cycle* and *sequence* have over the other services. It is important to mention that types of control services with incoming thick lines are one order higher than other services at the outgoing end of a thick line, because they have these other services (both basic and control) as parameters.

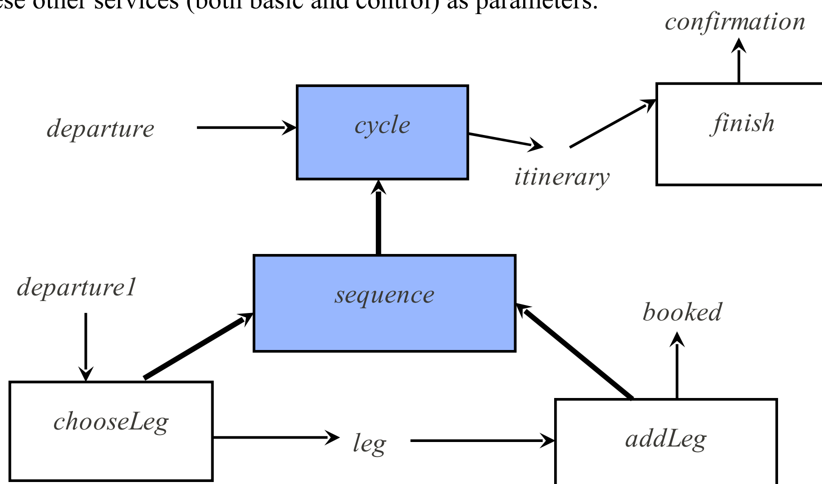

Figure 3. Higher-order workflow with data dependencies

At a closer look one sees that Figure 2 still does not show some important information – it does not 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 input-output descriptions. The data dependencies are shown in Figure 3, where the data entities: *departure*, *itinerary*, etc. are explicitly shown together with their roles as inputs and outputs of services. We have introduced the data entities *departure*, *leg*, *booked*, *departure1* (denoting the intermediate departure point of the travel), *itinerary* and *confirmation*. Their meaning should be rather obvious. 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, one can drop the explicit ordering control information. This is what we are going to use in composition process – derive control information from data dependencies. A particular type of control node is defined by the label of the control node of HOW and can be utilized for choosing of appropriate construct of practical business process language. We provide a precise semantics to our HOW, in particular, define its logical semantics. Representation in logic has the advantage that it allows to reason in a very precise way about the workflow. It enables us both to generate HOW of a composite service and to reason about the reachability of goals in workflow models. General relations between logical presentation, HOW and process languages are shown in Figure 4.

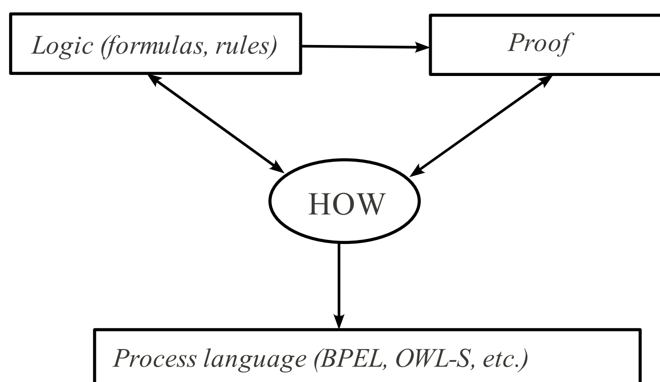

Figure 4. Logic, HOW and process languages

Services, that is, nodes, in HOW (both basic and control) can be presented as formulas in a particular logic (see the next section). Proofs in the logic can be transformed into HOW (and vice versa) that allows one to synthesize a new HOW from a proof. At the same time, HOW can be transformed into business process languages via instantiating control nodes by the language constructs, which allows one to generate composite processes (composite services) automatically. We are going to use logic for defining the semantics of workflow.

5. Representing HOW and Service Models in Logic

One can use intuitionistic propositional logic for representing the semantics of workflow. For example, the functionality of services *chooseLeg* and *addLeg* is

representable by the following implications where the data names have the role of propositions:

departure1 ⊃ *leg*; *leg* ⊃ *booked*

The first tells us that having *departure1* one can obtain leg. The second tells us that having *leg* one can obtain *booked*. 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, and in our case it can be the data item that is the respective input or output of the service. According to the semantics of intuitionistic logic, the implications have a computational meaning, or in other words – their realizations must be functions. In our case they are the services *chooseLeg*, *bookLeg* and *finish*. They can be shown in logical style as the realizations of the formulas written after colon:

departure1 ⊃ *leg : chooseLeg*; *leg* ⊃ *booked : bookLeg itinerary* ⊃ *confirmation : finish*

Now we are going to the representing of control. The difference between a basic service and a control node is that the control node, in addition to simple data inputs and outputs, has other services as parameters. However, we do not want to operate in logic with names of services (this would require the usage of the higher order logic), but with logical propositions only. Therefore we describe the services that are parameters by means of their pre- and postconditions, binding them with control nodes. This is denoted by dashed arrows and logically this is represented by a nested implication. Now we can change the notation and use no thick arrows at all, but only thin arrows that should have a special marking (dashed arrows) for inputs and outputs of subtasks. This allows us to use uniform representation and binding data for all workflow models as shown in Figure 5 for the travel example. In this figure, *departure1* and *booked* express pre- and postcondition of a node (or of a sequence) that is a parameter to the *cycle*.

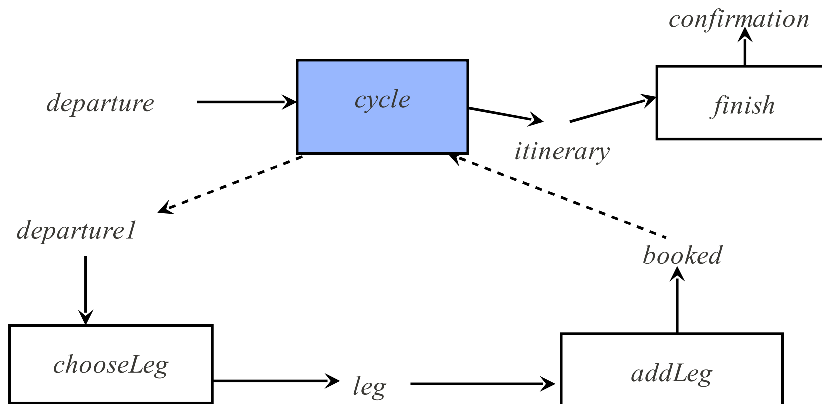

Figure 5. Expressing higher-order workflow by means of data dependencies

The logic behind this workflow is expressed by four formulas, one formula for each service and control node in the workflow. Let us look now at the control node *cycle*. It has a precondition *departure* and a postconditon *itinerary*. Besides that, it produces a precondition *departure1* for the sequence of services that constitute the body of the cycle. This precondition expresses the availability of current departure (for the current leg of the trip) and requires a postcondition *booked* of the sequence. The

postcondition *booked* expresses the availability of a partial itinerary that has the newly found leg as its last part. All this is expressed in the logic as follows:

departure∧*(departure1*⊃*booked)* ⊃ *itinerary:* `cycle.`

Indeed, the implication *departure1* ⊃ *booked* represents the functional input realized by a sequence of services, hence it must be on the input side of the main implication for cycle. It is important to understand that the implication *departure1* ⊃ booked represents an input of the control node *cycle*, but its value is not a data! It is body of the loop performed by the control node *cycle*. Here the higher order is hidden now. This body must be synthesized from other available services, and we call such implication on the left side of another implication a subtask. All other nodes are described by simple implications:

departure1 ⊃ *leg :* `chooseLeg;`

leg ⊃ *booked :* `bookLeg;`

itinerary ⊃ *confirmation:* `finish;`

The fragment of logic used here is already sufficient for representing dataflow and also synthesizing services as nested flows of services taking into account their data dependencies. This has been described in detail in our paper [3].

Looking at the workflow representation by means of data dependencies (e.g., Figure 5), we can see that this corresponds exactly to a bipartite graph that represents a service model. A service is represented by a node and has connections to its inputs and outputs. A higher-order service is represented by a node that has connections to its inputs and outputs as well as to inputs and outputs of its subtask. The top level model in a hierarchical service model has components that are service models themselves. They can be unfolded in the HOW representation. As a consequence, the representation of HOW in logic is applicable to service models as well.

6. Specification Language for Service Composition

In this section we will describe a specification language for the service composition. This language allows to specify the expressions of the logic we described in Section 5 and is implemented in the model-based software development platform CoCoViLa[3]. Examples of HOW and hierarchical models created in CoCoViLa will be given in the next section.

Specification language includes the following constructions:

- Specification variables: `type id, [id2, ...];`
- Variable binding: `variable1 = variable2;`
  Variable binding specifies the equality between specification variables `variable1` and `variable2.`
- Axioms: `input -> output {realization};`
  Axiom's input may include multiple variables, corresponding to the propositional variables of intuitionistic logic, separated by commas denoting conjunctions. Input may also include subtasks:
  `[a->b], c -> output{realization};.`
  In this example, axiom's input contains a subtask `[a->b]`, and a variable `c`.
- Aliases: **`alias`** `aliasName=(variableName1, ..., variableNameN);`

3 CoCoViLa – http://www.cs.ioc.ee/cocovila

Alias variable allows to define a structure containing several variables.

- Wildcards: **`alias`** `aliasName=(*.variableName);` In the case of wildcards alias structure will contain all the variables having a name *variableName* from the same specification.
- Control variables: **`void`** `controlVariableName;.` Control variables are declared with the type void and are used to specify the order of execution of methods.
- Specification language supports inheritance.

7. Specification of HOW and Hierarchical Components

In this section we will give examples of the HOW and hierarchical components. These examples use the specification language we described in Section 6 and are implemented in a Java-based visual modelling tool CoCoViLa that offers structural program synthesis and is able to handle specifications given in either textual or visual form. Visual components can have inputs and outputs and fields that are variables in the specification language. Input and output variables can be connected by creating a variable binding (data dependency relation). Values to the fields are given through the properties window, that can be opened by double-clicking on the component.

In the first example we will describe how the services and data entities are modelled and specified in our approach. Second example shows higher-order services and language extensibility possibilities. Third example in this section shows how to specify and model hierarchical components.

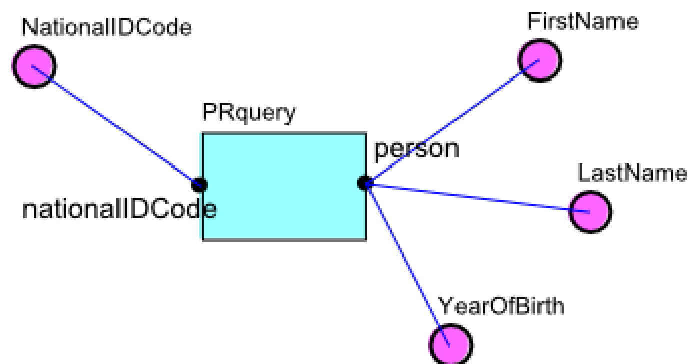

Figure 6: Visual representation of the service computation problem.

7.1. Example 1

Suppose we have a service *PRquery* that takes person's national ID-code (*NationalIDCode*) as input and returns a data structure called *person*. This structure contains the following fields – *firstName*, *lastName* and *yearOfBirth*. Problem's visual representation is given in Figure 6. Rectangle in the figure represents the service, that has input *nationalIDCode* on the left – connected to the data entity *NationalIDCode* and output variable *person* on the right – connected to the data entities *FirstName*, *LastName* and *YearOfBirth*. Specification for the service *PRquery* is the following:

```
String nationalIDCode;
String firstName;
```

```
String lastName;
int yearOfBirth;
alias person = (firstName, lastName, yearOfBirth);
nationalIDCode -> person {prquery};
```

where *{prquery}* is the Java method that realizes the axiom:

```
nationalIDCode -> person,
```

nationalIDCode and *person* are input and output variables. Note that the structure *person* can hold the fields with different types. The whole problem of computing, for example, *LastName* and *YearOfBirth* from the *NationalIDCode*, is specified as follows:

```
NationalIDCode NationalIDCode;
PRquery PRquery;
FirstName FirstName;
LastName LastName;
YearOfBirth YearOfBirth;
PRquery.nationalIDCode = NationalIDCode.value;
FirstName.person = PRquery.person;
LastName.person = PRquery.person;
YearOfBirth.person = Prquery.person;
NationalIDCode.value->LastName.lastName,
                              YearOfBirth.yearOfBirth;
```

Connections on the model are specified as equations and our goal to compute *FirstName* from *NationalIDCode* is given in the last line. The notation *LastName.lastName* allows us to take the field *lastName* out of the structure *person* specified in the data entity *LastName*.

7.2. Example 2

We will now continue the example and show how to use higher-order services. We had a service that given a *NationalIDCode*, returned *FirstName*, *LastName* and *YearOfBirth*. Let us have a list of *NationalIDCode*-s and a goal to find *FirstName*-s and *LastName*-s that correspond to those *NationalIDCodes*-s. To achieve this, we can add a higher-order service that takes the list of *NationalIDCode*-s and calls the service *PRquery* for each of them (i.e., *PRquery* is subtask of the service *Cycle*). The new service can be called *Cycle*. It returns a list of first name and last name pairs. We can extend our model even more, and add a custom service that visualizes results from the *Cycle* service in the Java table. Figure 7 shows the model containing *Cycle*, its subtask *PRquery* and a service *Table*. In addition we have included a resulting Java table and a properties window for a *Table* service, that enables to give values to the fields in the specification.

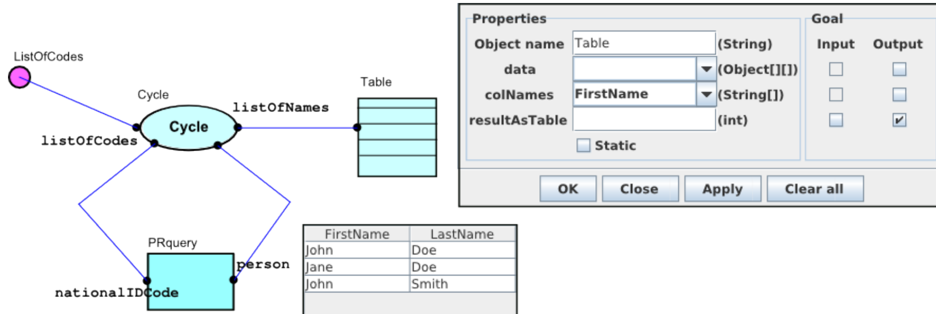

Figure 7: Higher-order service Cycle, and Table service with its properties window and results of the problem visualized in Java table.

Specification for the *Cycle* service is the following:

```
String[] listOfCodes;
Object[][] listOfNames;
String nationalIDCode;
String firstName;
String lastName;
int yearOfBirth;
alias person = (firstName, lastName, yearOfBirth);
alias name = (firstName, lastName);
listOfCodes,[nationalIDCode->person]->
                                    listOfNames {cycleCodes};
```

The last line in the specification says that in order to get the *listOfNames*, two inputs need to exist – *listOfCodes* and a subtask *nationalIDCode -> person.* Subtask *nationalIDCode -> person* is of course relalized by *PRquery* that we described in our previous example.

One of the benefits of our approach, is that we can extend the composition modelling language with custom services. To demonstrate that we have created a *Table* service, that takes an input data and visualizes the results as a Java table. This service is also usable with other data types and data-structures. Column names can be specified trough the textual specification or through the properties window of the *Table* service. Specification for *Table* service is the following:

```
Object[][] data;
String[] colNames;
int resultAsTable;
data, colNames -> resultAsTable{showTable};
```

where *showTable* is a Java method that given a data and column names, visualizes this data in the Java table.

If we give some values to the *listOfCodes* variable and set *Table.resultAsTable* as an output, we can try to solve a goal: *listOfCodes.data -> Table.resultAsTable.* If the problem is solvable we will get an algorithm of the solution. A Java code is generated from the algorithm, and after the execution of Java code a result is obtained as a table.

7.3. Example 3

The whole model shown in Figure 7, can now be saved as a hierarchical component and used in the model. This component could for example have two input variables and two output variables. Figure 8 shows a new hierarchical component the left and an

original model it represents on the right. When using the component user can choose whether to insert a list of codes and get a result as a table or insert just one ID-Code and get a structure *person* as a result.

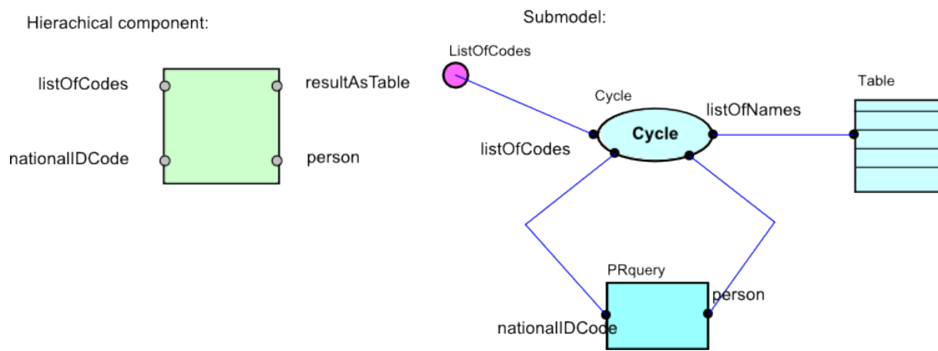

Figure 8: Hierarchical component (on the left) and the model it contains (on the right).

Specification for the new component is the following:

```
PRquery PRquery;
Cycle Cycle;
Table Table;
Table.colNames = {"FirstName", "LastName"};
ListOfCodes ListOfCodes;
PRquery.nationalIDCode = Cycle.nationalIDCode;
PRquery.person = Cycle.person;
Table.data = Cycle.listOfNames;
Cycle.listOfCodes = ListOfCodes.listOfCodes;
```

All the complexity and description of subtasks is now hidden in the lower layer (in components). We will look at some more complex examples of hierarchical models in Section 8, where hierarchical models are built for Estonian e-government information system.

8. Public Services of Estonian e-government

In this section we describe experiments that were done with services of the Estonian e-government information system that has a service oriented architecture. Central part of Estonian e-goverment information system is the X-Road data-exchange layer [2]. Estonian e-government services enable to read and write data to and form the national databases, develop business logic based on data, etc. In X-Road data exchange is handled by SOAP messages, Web services are described in WSDL and service descriptions are published in a UDDI repository.

Currently queries to the national databases have to be done one-by-one and possible semantic connections between services are ignored. Estonian e-government information system integrates more than 2000 services from abount 100 different organisations. Domain of this size is quite hard to handle entirely manually. In this work we refer to the whole Estonian e-government information system as X-Road.

Each organisation that has joined X-Road operates one or more registries or databases. Data in these registries can be accessed or modified through the web services that are described with WSDL descriptions. We will demonstrate our tool on the model that is created from five annotated WSDL descriptions – business registry, population registry, vehicle registry, migration registry and pension insurance registry. The process of creating service models from annotated WSDL descriptions is automatic, but currently X-Road service descriptions lack the annotations, therefore, it is hard to test our method on the whole X-Road.

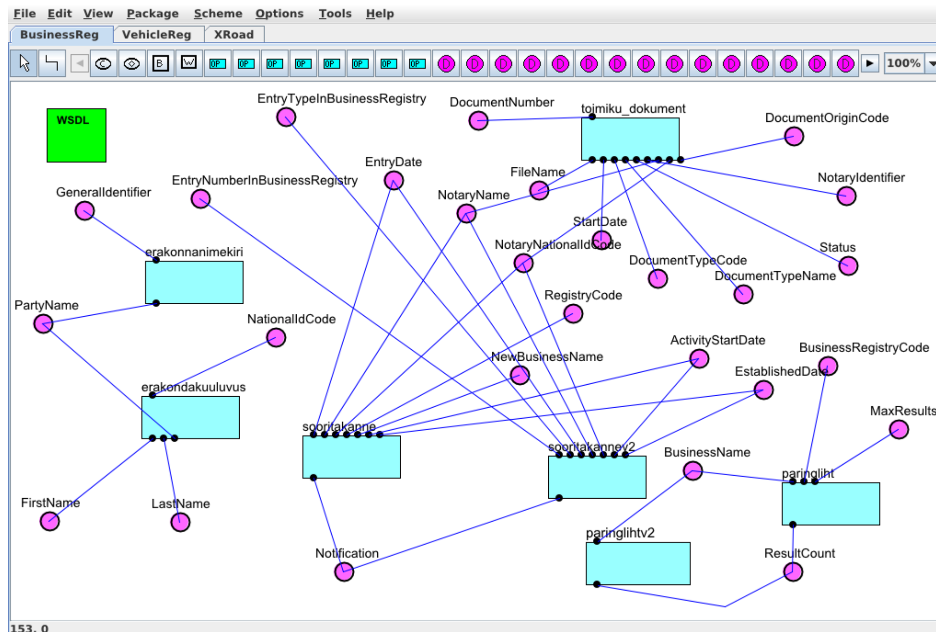

Figure 9: Fragment of the business registry's service model.

Figure 9 shows a fragment of the business registry service model. As in previous examples, rectangles represent services and circles represent data entities. WSDL component in the top left corner contains additional information extracted from the service description about the particular registry. For readability we have simplified the service model and show only 7 atomic services out of the 42 available for the business registry. Service models for other registries are similar.

All service models can be saved as hierarchical components and used in hierarchical models. We can select which inputs and outputs to show on the hierarchical component. Hierarchical model of X-Road could represent the whole X-Road, but in this case the number of hierarchical components would be more then 100, which would be rather hard to handle in a single model. In addition X-Road has quite complex access controls. Although, our 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 the models. Fortunately we do not have to put all components into one model and can create different views for users with different expertise and access rights.

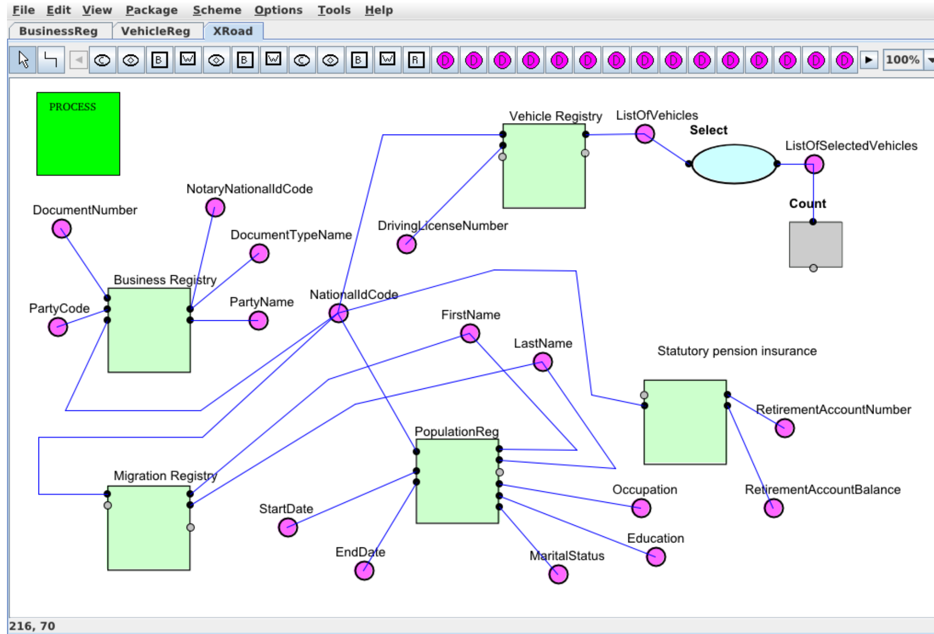

Figure 10: Hierarchical view of the Estonian e-government model.

Figure 10 shows an hierarchical view of the X-Road model. Rectangles with inputs and outputs are hierarchical components that contain submodels and circles represent data entities that are used for visualization. The submodel for business registry was shown in Figure 9. In addition, there are hierarchical components for migration regitry, vehicle registry, population registry and statutory pension insurance registry. Information in the submodels (including higher-order services) is used in the synthesis but their complexity is hidden from the user. We have only brought out data entities that might be interesting to the potential user of the model.

PROCESS component in the top left corner in Figure 10, contains additional information about the compound service. In the synthesis process this component will also gather information about the services included into the workflow or propagate information to the model. It is implemented as a superclass of the scheme.

Previously we have experimented with the generation of complex service description in BPEL and OWL-S [4], where each service component that was executed wrote out its own description and the BPEL or OWL-S component took care of the process description. This approach works well with flat, single layer models, but it gets complicated with hierarchical models and with custom components that extend the default workflow language. However, synthesis process will still generate a structure of the compound service and we plan to add a possibility to translate this structure into BPEL in the future.

This model includes also customized *Select* and *Count* services, shown in the top right corner. These services will be used in the example and we will then explain them in more detail.

8.2. Example

Let us now consider the case where we need to discover notary's home address and information about the vehicles the notary owns that are newer than 10 years, from a given document number. The main goal is then:

DocumentNumber -> Address, NumberOfVehicles.

We expect that the domain expert knows that person's home address can be obtained from *NationalIDCode.* From the *DocumentNumber* we will, however, get a *NotaryNationalIDCode,* it is obvious that in this context *NotaryNationalIDCode* and *NationalIDCode* have the same meaning. Therefore, we can extend our specification with the fact:

```
BusinessReg.NotaryNationalIdCode = NationalIdCode.data.
```

Vehicle information comes from the Vehicle registry and again we need *NationalIDCode* for input. However we do not get the answer for the *NumberOfVehicles* newer than ten years from the vehicle registry. Instead of this, Vehicle registry has a service that outputs a list of vehicles a person owns. This means that additional components need to be added to the model, that are able to:

1. Select the elements of the list that correspond to some condition. We call this component *Select.*
2. Return the size of the list. We call this component *Count*.

Specification for the *Select* component is the following:

```
String[][] listOfVehicles;
Object[][] selectedVehicles;
String condition;
listOfVehicles, condition -> selectedVehicles {select};
```

And specification for the *Count* component is the following:

```
Object[] inputList;
int size;
inputList -> size {count};
```

The actual goal that will be specified on the model to compose the compound service is then:

```
DocumentNumber.data ->  PopulationReg.address, Count.size.
```

And the order in which atomic services are added into the composition are:

```
documentNumber->notaryNationalIDCode {toimiku_dokument}; (BusinessReg)
nationalIDCode->rr57Output {RR5}; (PopulationReg)
nationalIDCode->listOfVehicles{paring22}; (VehicleReg)
listOfVehicles->listOfSelectedVehicles {select}; (Main)
listOfSelectedVehicles->size {count}; (Main)
```

Text in the bold indicates the model to which the service belongs to. Last two services belong the the main model, that was shown in Figure 10. Note that the *address* structure is automatically extracted out of the output variable *rr57Output*. User does not need to know anything about the exact services in the submodels and can only work on the hierarchical level.

9. Related Work

Logic-based approach to automating service composition has been proposed a number of times by the research community. In the context of this work we are interested in service composition methods and tools that automate workflow generation and are suitable for search from large number of services. Unfortunately, little has been written about the scalability of composition methods, and no large examples are given except [4].

In the approach taken by the following four tools, some predefined pool of services exists (for example, in a service registry), and a goal that has to be satisfied by the compound service is defined by the user. Solution that satisfies the goal is constructed automatically by a planning method that is using the actual service descriptions as the planning data. Our approach differs from these approaches, because we have a possibility to change the specification obtained from a visual model. In addition, we can add new facts about the composition domain just before starting a synthesis process.

Sword [5] is a set of tools for composition and execution of web services. A service is represented by a rule expressing that given certain inputs, the service is capable of producing particular outputs. A rule-based expert system is then used to automatically determine whether a desired composite service can be constructed using existing services. If it is possible to construct such service, a plan for creating a service is constructed. Execution of the plan instantiates the composite service. Individual services are defined by their inputs and outputs. For each service, a rule is defined.

Rao et al. [6] describe a system based on propositional linear logic that allows to describe both functional and non-functional properties of services. DAML-S is used to represent semantic web services. The pool of DAML-S specifications are translated into linear logic axioms. Linear logic theorem prover is used to prove whether the user's defined goal can be achieved by composition of available atomic services. If so, the process model is extracted from the proof. Process model of the compound service can be translated into DAML-S or BPEL. Graphical user interface is used to visualise services.

Kona et al. [7] propose a method that takes a repository of semantically annotated services and a query and finds a directed acyclic graph that represents a compound service. Resulting composition graph can be translated into OWL-S. Prototype implementation has been tested in the bioinformatics domain. This method seems to be well scalable.

Haav et al. [8] propose a framework for informational web services of X-Road. This framework has two parts – annotation part and a logic-based composition part. Annotated services and a user request are translated into the tool's inner logical language that is then used for program synthesis in the theorem prover RqlGandalf. Result of the program synthesis is extracted as a Python program.

There are also tools that combine automated workflow generation with manual workflow construction. This is very useful with higher-order components, because, unless very detailed and complex specification language exists, a developer might do a better job than a planner. Following three tools fall into this category.

JOpera [9] 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 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 [10] is a service composition and execution prototype tool for OWL-S that has two basic components: a composer and an inference engine. Information about known services is stored in the tool's knowledge base that is used by inference engine to find matching services. The inference engine is an OWL reasoner written in Prolog and it is used to find available choices at each composition step. User starts the composition of the new complex service by selecting one of the services registered by the engine, to be the last service in the workflow. The rest of the workflow is created by selecting input services amongst those presented by the tool. This means that new services providing appropriate input data for already selected services 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, type, etc.) is supported.

Synthy [11] is a prototype of the semantic 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 service. Relevant services to fulfil the tasks are then discovered automatically by the system amongst available semantically annotated services. If there are no exact matches, services that could fulfil the task are discovered and a control flow is created automatically between those services. Resulting workflow is described in BPEL.

Difference with our approach is that automated workflow generation in JOpera is relaying on matching input and output values with the same name. In Web Service Composer matching input services are discovered automatically, but if there are more then one matching services, user still has to make a chose manually. Synthy is most similar to our approach, but it is still for the development of one workflow at the time, while in our approach we can change the resulting workflow simply by changing the goal.

There are not that few tools that can be applied to large sets of services. Hierarchical composition of workflows has also been implemented in Sedna [12] – visual webservice composition environment for scientists. Sedna supports only manual composition of services, but its visual language has been extended for supporting scalability and to decrease 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. Sedna allows to describe subworkflows first, and then to use them by other workflows. Subworkflows can be described by a WSDL that enables other workflows to invoke a subworkflow like a service.

10. Conclusion

Experiments with synthesis of services on hierarchical service models of the Estonian e-government information system have shown the feasibility of the approach where the services are divided in separate service parks and described by several ontologies. Technically this approach works well, one can handle large amounts (thousands) of

atomic services in this way. Especially the question of interoperability can be solved easier in a service park than in a general case. Still a problem remains with updating the models of service parks and of the user service model, because the changes in services are unavoidable, and the information about the changes must be passed to the persons responsible for the correctness of service models. We have not touched the problem of security, but information assurance problem is important, especially, in military applications. Service models can be in principle extended with additional information about the confidentiality and access rights. This interesting question remains for the future research.

References

[1] C. Petrie and C. Bussler, "The Myth of Open Web Services: The Rise of the Service Parks," IEEE Internet Computing, vol. 12, no. 3, pp. 96–95, 2008.

[2] 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.

[3] M. Matskin, R. Maigre, and E. 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.

[4] R. Maigre, P. Küngas, M. Matskin, and E. 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/

[5] S. Ponnekanti and A. Fox, "Sword: A Developer Toolkit for Web Service Composition," in Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI, 2002.

[6] J. Rao, P. Küngas, and M. Matskin, "Composition of Semantic Web Services Using Linear Logic Theorem Proving," Information Systems, Special Issue on the Semantic Web and Web Services, vol. 31, no. 4-5, pp. 340–360, 2006.

[7] S. Kona, A. Bansal, M. B. Blake, and G. Gupta, "Generalized semantics-based service composition," in IEEE International Conference on Web Services. Washington, DC, USA: IEEE Computer Society, 2008, pp. 219–227.

[8] H-M. Haav, T. Tammet, V. Kadarpik, K. Kindel and M. Kääramees, "A Semantic-Based Web Service Composition Framework", Advances in Information Systems Development, Springer US, 2008, pp. 379-391.

[9] C. Pautasso and G. Alonso, "JOpera: a Toolkit for Efficient Visual Composition of Web Services," Int. J. Electron. Commerce, vol. 9, no. 2, pp. 107–141, 04-5.

[10] E. Sirin, J. Hendler, and B. Parsia, "Semi-Automatic Composition of Web Services Using Semantic Descriptions," in Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003, 2002, pp. 17–24.

[11] V. Agarwal, G. Chafle, K. Dasgupta, N. Karnik, A. Kumar, S. Mittal, and B. Srivastava, "Synthy: a System for End to End Composition of Web Services," Web Semantics: Science, Services and Agents on the World Wide Web, vol. 3, no. 4, pp. 311–339, 2005.

[12] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, and J. Patel, "Sedna: a BPEL-Based Environment for Visual Scientific Workflow Modelling," in Workflows for eScience – Scientific Workflows for Grids, I. Taylor, E. Deelman, D. Gannon, and M. Shields, Eds. Springer Verlag, 2007.