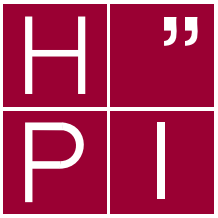
HASSO-PLATTNER-INSTITUT
für Softwaresystemtechnik an der Universität Potsdam



Survey on Service Composition

Dominik Kuropka
Harald Meyer

Technische Berichte Nr. 10
des Hasso-Plattner-Instituts
für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik
an der Universität Potsdam

Nr. 10

Survey on Service Composition

Dominik Kuropka
Harald Meyer

Potsdam 2005

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Die Reihe *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam* erscheint aperiodisch.

Herausgeber: Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Redaktion: Dominik Kuropka; Harald Meyer
Email: dominik.kuropka; harald.meyer}@.hpi.uni-potsdam.de

Vertrieb: Universitätsverlag Potsdam
Postfach 60 15 53
14415 Potsdam
Fon +49 (0) 331 977 4517
Fax +49 (0) 331 977 4625
e-mail: ubpub@uni-potsdam.de
http://info.ub.uni-potsdam.de/verlag.htm

Druck: allprintmedia gmbH
Blomberger Weg 6a
13437 Berlin
email: info@allprint-media.de

Heft 10 (2005)
ISBN 3-937786-78-3
ISSN 1613-5652

Survey on Service Composition

Dominik Kuropka and Harald Meyer

2005-10-21

Executive Summary

It is predicted that Service-oriented Architectures (SOA) will have a high impact on future electronic business and markets. Services will provide an self-contained and standardised interface towards business and are considered as the future platform for business-to-business and business-to-consumer trades. Founded by the complexity of real world business scenarios a huge need for an easy, flexible and automated creation and enactment of service compositions is observed.

This survey explores the relationship of service composition with workflow management—a technology/concept already in use in many business environments. The similarities between the both and the key differences between them are elaborated. Furthermore methods for composition of services ranging from manual, semi- to full-automated composition are sketched. This survey concludes that current tools for service composition are in an immature state and that there is still much research to do before service composition can be used easily and conveniently in real world scenarios. However, since automated service composition is a key enabler for the full potential of Service-oriented Architectures, further research on this field is imperative. This survey closes with a formal sample scenario presented in appendix A to give the reader an impression on how full-automated service composition works.

Contents

List of Figures

1. Introduction

The main objective of the Adaptive Services Grid (ASG) project[1] is the development of an open and generic software platform for discovery, creation, composition and enactment of services. The aim of this deliverable is the presentation of current efforts for the automated composition of services. Since the idea of service composition is derived from workflow planning we will start in section 1.1 with a brief overview on the roots of workflows and workflow planning and their relation to service composition. Section 1.2 discuss service composition in the context of the ASG project. The two main parts of this deliverable are chapter 2 and 3. The first one presents the state-of-the-art on (manual) planning of business workflows and (manual) service composition. The second one presents current research efforts in the field of semi-automated and full-automated service composition. The deliverable is closed with a conclusion and outlook in chapter 4. Appendix A contains a formal sample for some services including their specification. Furthermore a sketch is presented, on how an automated service composition including matchmaking could be derived from a given sample user request.

1.1. Historical overview on Workflows and Service Composition

Workflow management is the (technological) answer for the demand on scheduling and supervision of administrative business processes. This demand emerged in the 1990s by the wide application of Business Process Re-engineering (BPR) in various business domains. Due to the increased complexity of business and administration the limits of dividing tasks up by functional division become apparent. For this reason processes and especially their re-engineering was put on the agenda of most executives at those time.

Previous to the focus on processes and BPR the traditional solution for complexity was the 'divide and conquer' approach with focus on functionality of tasks. This means that complex tasks where divided in several sub-tasks which where handled by specialists. Over time the complexity of subtasks raised because of new legal rules, new business models or new functionality due to technological advancements. This lead to the situation that subtasks where divided into smaller sub-tasks again which where handled by even more specialist people. After this dividing has been repeated several times, following challenges emerged:

- *Slow process execution*: It was observed that the passing of business cases (e.g. a sales orders or an insurance claims) through the organisation was slow and takes a long time although the

[1]This paper presents results of the Adaptive Services Grid (ASG) project (contract number 004617, call identifier FP6-2003-IST-2) funded by the Sixth Framework Programme of the European Commission.

summarised processing time of all tasks was relatively short. The main problem was, that the flow of a business case through the organisation was fragmented into a lot of small tasks handled by different people. This caused a frequent transfer of documents, which delayed the processing of the case significantly. This was even worsened by the fact, that the transfer of business cases and relevant data from one resource to the other needed much time since it was not digitalised. Further, the assignment of resources was not flexible enough. Usually the assignment was done by some artificial criteria (e.g. by the first letter of the last name of the customer). This lead to the situation that some resources has been overloaded while others run idle. Finally, a dynamic re-assignment of resources was not possible in most cases, since nobody had a clue what was the current load on and availability of the resources. This leads to situations where already assigned cases where not processed for longer time because the processing staff where not available for example due to holidays or disease.

- *Loss of control on process execution*: Another experience made in the past was that in the functional division approach nobody was responsible for a specific business case as a whole. While every specialist executing a specific function is responsible for all business cases which are scheduled for him and which are on his desk, he loses this responsibility after successful execution of his task. If somebody wants to find out the current status of a business case, in the worst case he has to ask all employees to get the information who is currently working at the business case and to find out what is the status of the case. This makes the assurance of deadlines complicated and time intensive.

- *Missing overview on processes*: Finally a result of the functional division approach was, that each specialist had deep knowledge about his special field but nobody had the overview on the whole process. This led to inefficiencies due to lack of understanding on what a small change in task processing may cause in the subsequent tasks. Also cases are well known, where tasks get obsolete by time or technology but which are still executed because nobody is able to see that the result of the task is neither used by any other task nor relevant for the business any more.

To avoid the mentioned problems business realised in 1990s that the modelling and enactment of business processes as a whole is needed to cope with the new challenges on flexibility, complexity and efficiency. Parallel to this Workflow Management Systems (WfMS) emerged as the technology for automated support on process enactment. These systems are able to control, coordinate, and monitor all aspects of a process activity: function, data, resource and embedding into the whole process. To enable this automated support by WfMS business processes are translated and detailed into a formal and computer-readable process model, which is usually named *workflow*.

With the advent of the Service-oriented Architectures (SOA) [Bur00] a new field for the application of WfMS evolved. SOA is about structuring of software by using a service-paradigm. Thus software and services as well as their properties and their relationships to other objects like service providers and requestors are in focus of SOA. Services are self-contained functions which accept requests and return responses through a well-defined interface. Especially in business contexts embedding of services into existing processes is crucial. Furthermore new business models are enabled by SOA which often include the need for composition of services to independent processes. For this

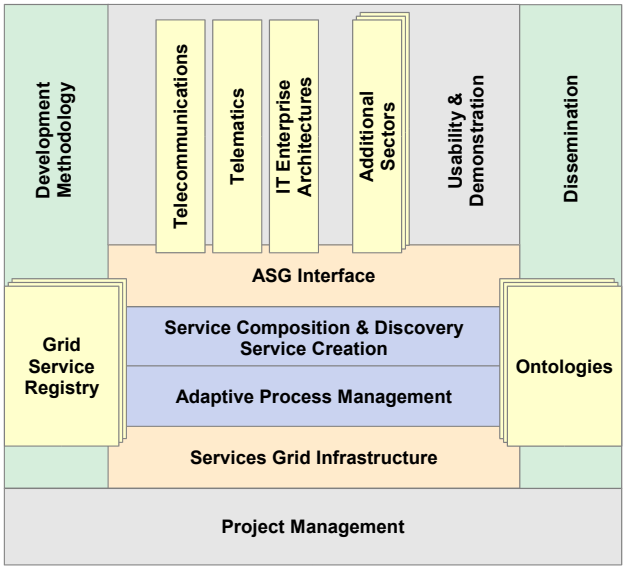


Figure 1.: Structure of the ASG project.

reason it makes sense to reuse and adapt existing tools, like WfMS to support the task of controlling and enactment of service compositions. Like the enactment of conventional processes, the enactment of service compositions has to deal with data and it's proper transfer as well as with resource management in the sense of choosing the right provider in case a service is provided by several ones. In chapter 2 we start with the illustration of the state-of-the-art of manual workflow planning and manual service composition in general and focus later in chapter 3 on current research efforts on automated service composition in particular, since the latter one is—as shown in the next section—of direct relevance for the ASG project.

1.2. Positioning in the ASG Context

The composition of services is an important—though challenging—feature of the ASG platform, since it is unrealistic to assume that all user request can be satisfied by execution of exactly one existing service. The idea is that user requests are not specified as direct service calls, but rather as a semantic description of the current situation (state and data) and the intended goal of the user. On the basis of this description a service composition planner creates a valid service composition which is able to fulfil the indented goal. Service composition increases the value of the ASG platform since it enlarges the range of possible solutions that may be presented to the user. For this reason the service composition planner (as part of *Service Composition & Discovery*) has a central role in the ASG project which is also reflected by it's central position in the ASG project structure (refer to figure 1).

Figure 2 presents the ASG subsystems and their dependencies. Technically is service composition planner encapsulated in an own subsystem named *composition*. It is directly controlled by

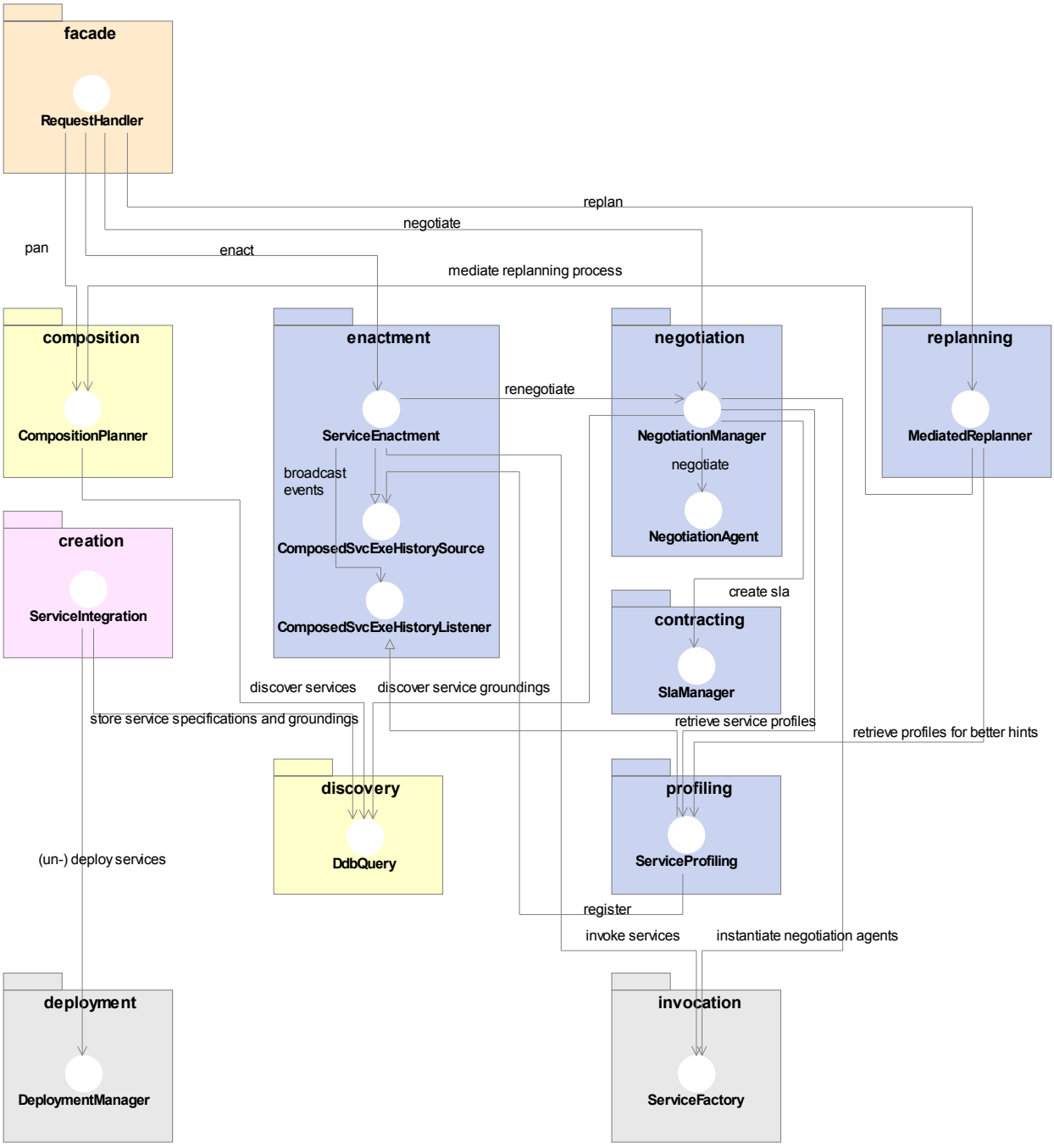


Figure 2.: ASG subsystems and interface dependencies.

the *facade* subsystem which provides the *composition planner* with user requests. The *composition planner* itself accesses the *deductive database query* interface to build up a service compositions. Starting from the given input data the composition planner finds proper services having a specification matching to the given data and the output data of their respective predecessors. The final, resulting service composition, which is in fact a composition of service specifications, is returned to the *facade* subsystem which forwards them to the *enactment* subsystem. This subsystem initiates negotiation on services having an specification matching to the specifications in the composition and binds the most suitable one to the composition. After all services are bound, the composition is enacted and the *invocation* subsystem is used for the invocation of the services at their particular provider.

2. State-of-the-art

2.1. Manual Planning of Business Workflows

The starting point for a manual planning and modelling of business workflows is usually a given or planned technological and organisational structure of an enterprise. This structure is often described by a textual or graphical (e. g. using ARIS [Sch00]) but usually semi-formal description of business processes. So it is the task of a human workflow planner to concretise these business processes in that way, that they are precise enough to be executed full-automated by a workflow management system. To achieve this goal the workflow planner has first to ensure, that all relevant tasks as well as technological and organisational entities are modelled properly within the workflow management system. At least the following aspects of a task have to be modelled and stored in the system:

- *Constraints on (human) Resources* describe what abilities (e. g. speaks Spanish fluently) and competencies (e. g. is a purchasing agent with the accreditation to make contracts up to 10,000 €) the person in charge has to provide. These constraints are automatically evaluated during workflow execution by the workflow management system to ensure that a task appears only on the work list of appropriate persons. For a sound modelling of these constraints a model of all involved organisational entities is indispensable.

- *Input and Output Data* are described in order to enable a proper transfer of data from one task to an other during the enactment of workflows. This description includes a formal specification of the syntax of the data including used data types and structuring. Furthermore, an informal description of the semantics of data (meta data) can (and should be) included.

- *Technical Issues* subsume all other technical stuff which is needed to manage the automated enactment of a task. This usually includes glue code to access input and output data from databases or applications as well as code to execute proper applications or input masks on the users desktop. To solve the technical issues all involved technological entities have to be modelled and they have to provide adequate interfaces for access and manipulation of data or invocation of proper applications.

Having the tasks well defined, the workflow planner can start with the modelling of workflows. This includes the specification of logical and temporal dependencies on tasks usually named as *control-flow*. The control-flows defines under which conditions a task has to be executed or skipped. VAN DER AALST ET. AL. identifies in [vdAtHKB03] twenty[1] different patterns of control-flow types. Commonly observed examples of these patterns are simple sequences (e. g. task b is executed always after task a), parallel splits (e. g. tasks a and b are executed in parallel) or exclusive choices

[1] In the named paper.

(e. g. either task a or task b is executed). Next to control-flow the *data-flow* plays an important role in workflow modelling. The data-flow defines which output data of which task has to be delivered to which input data connector of a task. Although for humans data-flow seems to be obviously inferable from a given control-flow structure, it is sometimes not trivial to find a formal representation which is non-ambiguous in all possible cases of given data and the workflow enactment path resulting from the specified control-flow. It is a general practice to create workflows step by step from given business process descriptions by sequentially adding new tasks to a workflow and connecting them by data- and control-flow to the already modelled tasks. The challenge in this proceeding is to achieve a workflow which can be handled without human intervention by a workflow engine in ideally all possible situations. This means that ideally all eligible and exceptional situations have to be modelled properly. For this reason traditional workflow management systems are mainly used in stable and predictable environments like for example administration of insurance claims or general business administration with well defined processes. Since not all business environments are as stable as the aforesaid, research on making workflow management systems more flexible is still ongoing. Next to providing better methods for exception modelling and handling also approaches for adaptation of already running workflows are in discussion. An overview on different approaches increase flexibility of workflow management systems is presented in [RRD04].

To enable full-automated execution of workflows a machine understandable which usually means more or less formal description of workflows is needed. Over time a huge amount of workflow modelling languages have been developed. Languages like *Petri nets* [Pet66, Pet81] or *π-calculus* [Mil99] have a strong scientific and theoretical background. Their main focus is the study of process phenomenona and their behaviour. For this reason they are very formal and precise and are often used as an theoretical backbone for other languages. However modelling of real world problems is often laborious and inconvenient. The reasons for this valuation are for example that modelling of even relative simple real world workflows tend to result in complex representations. Furthermore, these languages focus purely on the modelling of data- and control-flow, usually without taking detailed task modelling and especially technical and organisational issues into account. For this reason vendors of workflow management systems invent their own workflow modelling languages or extended the scientific ones. They aim to find an optimum between the needed degree of formalism and the ease of the modelling of tasks and workflows, intuitive understanding of the models and proper tool support usually including visualisation. Just to name a few, well known languages of business workflow management systems are for example *Staffware* [Sta00], *COSA* [Sof99], *SAP R/3 Workflow* [SAP97] and naturally the relatively young *Business Process Modelling Notation (BPMN)* [BPN04].

2.2. Manual Service Composition

As already mentioned in section 1.1 are service compositions strongly related to workflows. For this reason experiences made in workflow planning are also relevant for the service composition task. Regarding some aspects service composition is easier and regarding other aspects it is more challenging than workflow planning. Service composition is easier regarding the aspect of task modelling, since the individual tasks of a service composition are service (operation) invocations

and services are described by their providers. This means that the major part of work regarding to task description is already done by the service providers. Service description languages like the *Web Services Description Language (WSDL)* [W3C01] ensure that a well defined service description includes information on the formal specification of the syntactical structure of input and output data (messages) of a service as well as information on technical issues (in particular invocation protocols etc.) needed to properly invoke the service. Furthermore, aspects regarding the resources and their constraints are not a part of a service description since it is the key concept of the Service Oriented Architecture (SOA) that the service requester does not need to take care about resources needed for the execution a service. This is the task of the service provider [Bur00]. However it has to be noted that the number of concrete services in the service landscape is higher than the number of tasks in usual workflow management systems, since service providers are not limited to one company. This also means that there might exist several different services having the capability to solve the same task probably with different costs and quality aspects making the decision on a concrete service not easy. Furthermore the dynamics of the service landscape, which include service change, deletion or creation, has to be taken into account.

Regarding the control-flow there is only a small difference between workflows and service compositions. There is no reason from the theoretical perspective why some control-flow constructs should not be allowed to appear in service compositions while they are allowed in workflows and vice versa. However it is possible that the frequency of specific control-flow constructs is different. Unlike for control-flows, the difference between workflows and service composition is clearly apparent for data-flows. Traditional workflow planning is done within one organisation using tasks and applications under the control of the organisation. This means that the semantics of tasks and especially their data is well understood, although it is not always trivial to connect the output of a task with the input of an other task. However there are experts within the organisation which can help the workflow planner on this job. Furthermore, mediation layers and application servers are often existing within the organisation since they are needed for the interoperability of different organisational parts, databases and applications. This infrastructures provide common view on tasks and data structures which can be reused for workflow planning. This situation is different in the context of the Service Oriented Architecture (SOA). A key idea of SOA is that service requestors and service providers can be and usually are from different organisations [Bur00] which are under certain circumstances residing in different parts of the world. Furthermore usual service description languages like WSDL [W3C01] contain only a syntactical description of services, their data and the invocation mechanisms. These circumstances hamper the task of the service composer, since he has to deal with heterogeneous technical environments. Especially the probability is high, that the output data structure of one service does not match the input data structure of an other service. This means that the service composer has to analyse the syntax of the different data structures and find a transformation mechanisms between them which matches their semantics. As a very simple example for illustration, it can be observed that the used format of dates often differs in different environments. The date '28^{th} of July 2005' can be represented as a string type for instance as '28.06.05' (German notation) or '2005-06-28' (ISO 8601). Alternatively it can be represented by a number type like 1119948614 (seconds since 1^{st} of January 1970, used internally by GNU/Unix systems) or as a well defined record like {day: 28, month: 6, year: 2005}. For this reason dedicated service composition languages like *Business Process Execution Language for Web Services (BPEL4WS)*

[ACD+03] provide special constructs which allow not only the representation of data-flows but also the specification of data transformations which are needed to adapt different data structures.

The general approach in modelling of service compositions is similar to modelling of workflows. Starting from a given informal process description, the human modeller has to find proper services which match directly or after some data transformation indirectly with the given initial data. Furthermore, a sequence of the services has to be found which are corresponding to the desired process goal. This composition of services is usually done step by step. Like in workflow planning, it is a challenge to find a service composition which can be handled automatically and without human intervention in ideally all possible situations. The probability of errors (especially communication failures) and changes of services in distributed environments is higher than in mono-organisational environments. Modelling of all these aspects (in case of using *BPEL* [ACD+03] as service composition language for example by using fault and compensation handlers) is a laborious challenge and it is impossible to ensure that all feasible and future situations are correctly modelled. Therefore the need for adaptive and fast approaches for modelling of service compositions is huge. If service compositions can be created cheaply, on demand and are enacted only one or few[2] times, then the probability for a change of the involved services is low. Furthermore, if an exception occurs it would be possible to re-compose a service composition with the aim that the new composition—with probably different services—works without failure. For this reason the semi- and full-automated service composition are interesting, since they promise an easy and cheap way to create new service compositions. An automated ad-hoc integration of new or improved services into existing business processes is a key enabler for a service market and it is only feasible with automated composition of services.

[2]Even if composition of services is cheap it will still need some time and computational resources to create a composition. For this reason it might be useful to store compositions for a shorter period of time to implement a caching mechanism. This might improve systems performance if a lot of requests for the same composition have to be enacted in a short time period. However this is only reasonable if the assumption holds, that the service landscape will not change significantly according the stored composition during the short time period.

3. Current Research Efforts

This chapter sketches some concepts and ideas of current research efforts on methods for semi- and full-automated composition of services.

3.1. Semi-automated Composition of Services

Composition of services is a step by step assembling of self-contained services to process with a specific capability as already described in chapter 2. Each assembling step consist at least of the following tasks:

- *Service Discovery* is the task of finding a service. Since services can be provided by various providers all over the world, finding of services is a non-trivial task. The challenges in finding proper services are the potentially huge number of service repositories as well as of different languages, law spaces, ontologies and business models. Searching for services on the world scale is the worst case according the named challenges.

- *Service Matchmaking* denotes the task of selecting a proper service from a list of existing services for assembling. Due to the potentially huge amount of services a matchmaking mechanism should have a high precision[1] in selecting a proper service. This is important to avoid overwhelming the modeller with unsuitable services. Furthermore, the variability of possible service capabilities is high due to the high complexity of the world and the various business scenarios. This also demands from a matchmaking mechanism a high recall, especially for rare and special services. High recall means, that ideally all matching services are found. In case capabilities of rare or special services are needed, high recall is important to ensure that at least one of the existing and matching services is retrieved.

- *Data-/Control-flow Linkage* defines the logical sequence of services and specifies the needed data transfers. Depending on the desired capability of the final service composition and the capabilities of the existing services the control-flow of the services may vary from a simple sequence to complex control structures including branches, parallelism and loops. Furthermore data-flow can be in the simplest case just an opaque propagation of the output of a service to the input of an other service. However in most cases data-flow will imply transformation of data and data structures to overcome different representations of the same or similar, but matching concepts.

[1] The term *precision* is used here in the sense common to the information retrieval domain. It is a measure of the number of relevant documents (in our case services) in the set of all documents (services) returned by a search.

Methods and tools which are able to cope with all above mentioned tasks are capable to provide a full-automated service composition of services and will be discussed in section 3.2. Since the provision of an automated data- and control-flow linkage is the hardest task and only solvable by using artificial intelligence techniques, we will discuss semi-automated service composition methods and tools first. Tools and methods for semi-automated service composition provide usually some automatisms for the service discovery and matchmaking tasks and leave the data- and control-flow linkage task to the user.

Service discovery can be automated by the provision of service repositories conceptually similar to the idea of white, yellow or green pages. The basic idea is that service providers register their services at the repository provider.[2] Furthermore these services should be described by a universal and standardised description to give the service consumer an opportunity to examine the service capabilities prior to the usage. A common standard on service descriptions and repositories is the *Universal Description, Discovery and Integration (UDDI)* [OAS02] provided by the Organization for the Advancement of Structured Information Standards (OASIS). Having a service repository in place, service discovery is done by inquiring the repository.

The automatisation of service matchmaking is a very important enabler for services application in real world scenarios. Since there are potentially many services out there, a mechanism is needed to find the proper one to solve a specific task or to assemble it into a service composition. Current approaches for service descriptions like WSDL and UDDI are lacking formal semantics. This has a negative impact on service matchmaking. A WSDL service description can only give very vague hints on the suitability of a service according to some given requirement on needed service capabilities. Indeed a WSDL description specifies the syntax of a service call. This includes information about how a service has to be invoked and what the structure of the input and output messages are, but it does not give any hints on the capability of the service and on the meaning of the input and output messages. In case the service composer wants to find a service which returns him the phone number of a person with a given name, it is not very helpful to know that there are many services existing which have a string-type as input and output. In fact for a successful service matchmaking the meaning of the inputs and outputs is relevant. In the above described sample case only these services with an string-type as input and output are relevant, that get a person name as input and return a phone number as output.

UDDI uses WSDL for the specification of the syntax of service calls, but UDDI also provides some natural-language service descriptions and some categorisation possibilities for service capabilities. This allows a partial automation of service matchmaking for service composition. Partial in the sense, that a computer is not able to find all possible matching services automatically without the help of the user. The reason for this limitation is the lack of formal semantics of service input and outputs which hinders an automated matching to previous or succeeding service output or inputs. However the computer can support the composition modeller in service matchmaking by visualisation of and search in categories as well as by providing keyword based searches on the textual service descriptions using information retrieval approaches [Kur04, BYRN99].

[2] An alternative is the usage of a crawler. The aim of the crawler is to sweep through the world-wide web and to store all service descriptions in the repository, which it finds at the home pages of the providers.

For a full automation of the matchmaking task, the whole semantics of services and their capabilities have to be described by formal, logical expressions. This can be achieved by the use of a semantic specification language like WSML [dB05] or OWL [MvH04] in combination with a service specification ontology like WSMO [RLK05] or OWL-S [Mar04] and a proper domain ontology. The semantic service specification language is the basic means of expression for the service description. It specifies which language constructs are syntactically correct and the abstract meaning of language constructs. For example a language might specify 'x OR y' as an expression meaning that either x or y has to be true, but it usually do not specify what x or y are or exactly mean. These kind of meaning of different entities and their relationships are defined by ontologies [Kur04, Smi02]. The service specification ontology provides the entities and relationships which are needed for a specification of services. It defines terms like 'service', 'operation' and 'capabilities' and their relationships among each other. Since services always have a relationship to a specific domain of the real world, the specification of their capabilities has to fall back on concepts of the specific domain. These domain specific concepts are provided by a domain ontology. Each semantic service specification is always related to at least one domain ontology. By help of the above described languages and ontologies the semantics of services and especially their preconditions and effects as well as the meaning of the input and output data can be expressed uniquely and exactly. This the matchmaking task can be ascribed as a logical reasoning or inference task. To be more concrete, the requirements on a service capability can be described by a logical expression which have to be logically matched against the logical service specifications. An sample service specification ontology can be found in appendix A.1; a sample domain ontology is presented in appendix A.2 and formal, semantic service specifications are shown in appendix A.3. An combined approach using information retrieval and semantic techniques is presented by CARDOSO and SHETH in [CS02]. They define next to pure syntactic similarity measures also operational and semantic similarity measures basing on semantic service signatures and ontologies. This allows an ordering of services according to their similarity towards the requested service signature specified by the user.

SIRIN, HANDLER and PARSIA present in their paper [SHP02] an example for a semi-automated service composition tool. Their tool uses OWL in combination with DAML-S (a predecessor of OWL-S) for the specification of services. As inference engine and service repository their tool uses an OWL reasoner built on Prolog for service discovery and matchmaking. The tool disburdens the task of service composition significantly by proposing the service composer proper services at every assembly step during the composition. The process of composing a service begins with the specification of the desired result. This desired result is specified by selecting concepts of the domain ontology which are representing the result. From this, the composition tool proposes a list of services which are matching toward this desired result in the sense that they are producing this result. In case there are several services producing the result, the user has to choose one of them by taking for example non-functional properties like execution cost etc. into account. In the next step the tool proposes other services which are capable of producing an output which is matching the input of the last, already assembled service. Again the user has to make a choice by non-functional properties in case several results are given. This activity is repeated until all needed inputs are given by the data the user knows to have and which can be used as input for the service composition.

In summary semi-automated service composition has to be evaluated as a useful step to ease the composition of services in general. However it does not provide any solution for the dynamics

of a web service world, since the service compositions are always driven by the human modeller. Regarding the adaptability of service compositions towards changes of the world the human factor is a bottle-neck, which rises the costs of individual service compositions. Therefore semi-automated composition of services is restricted to a relative stable and predictable domains with well defined providers and services. An automated ad-hoc integration of new or improved services into business, which is a key enabler for a service market, is not feasible.

3.2. Full-automated Composition of Services

Full-automated composition of services is necessary to exploit the whole potential of services. An ideal full-automated composition of services limits the task of a human to the specification of a initial state and a goal including optimisation criteria. Ideally it is the task of the system to find a proper service composition to meet the goal specified in the user request. Since an ideal full-automated composition of services removes the human factor as a bottle-neck, an on-demand composition for (in extreme case) each business case is possible. This also means, that each business case will automatically be adapted to the current state of the service world, reducing the probability of failure and potentially rising the efficiency by incorporating new, more efficient services into the business.

The initial state is a description of the given data or at least data-types as well as relevant world states in relation to the domain ontology. The initial state could be for example the information, that the user know the name of a person, which is coded in a record consisting of two strings, the first and the last name. Furthermore the request for service composition contains also a goal specifying what the result of the service composition should be. For example the goal could be that the user wants as a result a map with a cross marking the position of the person he knows the name of. It is also useful to add an optimisation criteria like cost minimisation. Appendix A.4 presents a formalised version of the above specified user request on basis of the domain ontology enlisted in appendix A.2. It is then the task of the full-automated service composer to find proper services and to assemble them to a valid service composition which is able to reach the user specified goal. Figure 9 on page 31 in the appendix shows exemplary how such a composition of services may look like according to the previously specified sample request.

Meteor-S is a prototypical platform for execution and enactment of semantic web services and processes developed by the university of Georgia. It supports the composition of web services, however their engine is limited to automatic binding of services according to user-defined constraints [AVMM04]. This means the user has to provide manually an abstract process containing restrictions on suitable services for each activity of the process. Prior to enactment of the services, these abstract processes are transformed to real processes by binding each activity to a real service. Thus this approach should not be labelled as a full-automated composition of services, it is a reasonable step towards an adhoc integration of new services into business. The specification of activities by constraints on services allows the selection of the most optimal service matching the constraints. However this approach is limited, since the general structure of the process remains untouched. It is for example not possible to replace two activities by a service which matches the sum of two constraints instead of each of the constraints alone.

Hierarchical Task Network (HTN) planning is an artificial intelligence (AI) planning method.

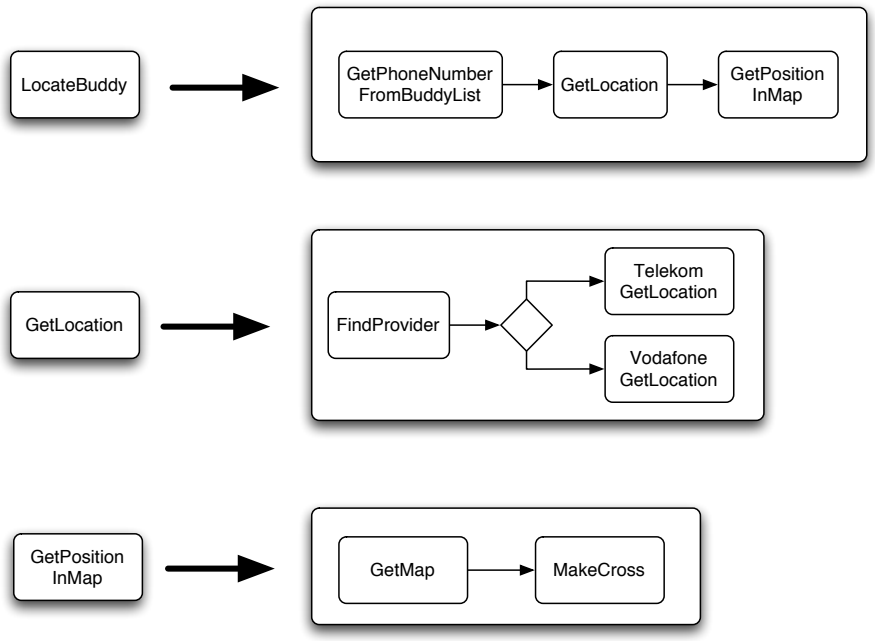


Figure 3.: Sample task definitions for HTN planning. (Source: [Mey04])

Each state of the world is represented by a set of atoms, and each action corresponds to a deterministic state transition. However, HTN planners differ from classical AI planners in what they plan for, and how they plan for it. The objective of an HTN planner is to produce a sequence of actions that perform some activity or task. The description of a planning domain includes a set of operators similar to those of classical planning, and also a set of methods describing how to decompose a task into sub-tasks (smaller tasks). Given a planning domain, the description of a planning problem will contain an initial state like that of classical planning—but instead of a goal formula, the problem specification will contain a partially ordered set of tasks to accomplish. Planning proceeds by using the methods to decompose tasks recursively into smaller and smaller sub-tasks, until the planner reaches primitive tasks that can be performed directly using the planning operators. For each non-primitive task, the planner chooses an applicable method, instantiates it to decompose the task into sub-tasks, and then chooses and instantiates methods to decompose the sub-tasks even further. If the plan later turns out to be infeasible, the planning system will need to backtrack and try other methods. [NAI+03]

Figure 3 shows an example for HTN task definitions. This definition consist of three tasks namely *LocateBuddy*, *GetLocation* and *GetPositionInMap* which are composed of several other tasks. On basis of this task definitions the possible result of a HTN planning is presented by figure 4. The initial state consist only of the task *LocateBuddy*. During the HTN planning this initial tasks is decomposed into smaller tasks until the lowest possible decomposition level is reached. This example shows also the limitations of the HTN planning method. Since tasks are manually defined as compositions out of other tasks the benefits of HTN planning in comparison to manual composition of an execution

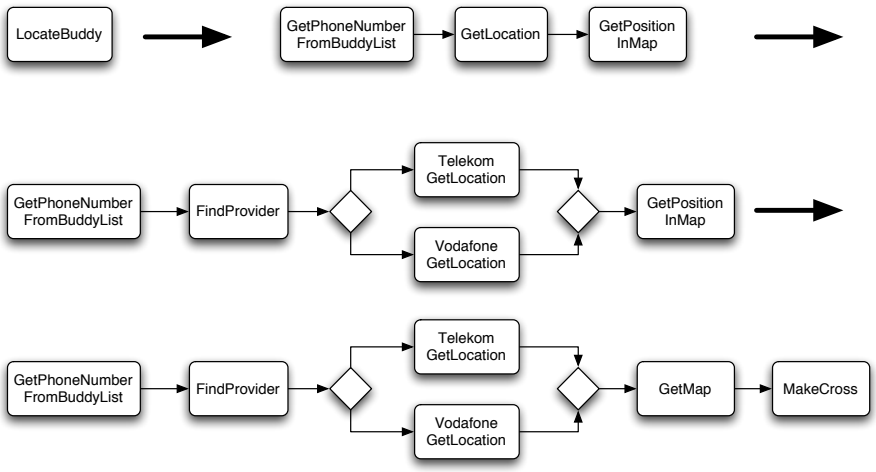


Figure 4.: HTN planning on task definitions from figure 3. (Source: [Mey04])

plan is low.

SIRIN ET AL. presents in [SPW+04] a prototypical implementation of a service composer which bases on the HTN-planner *SHOP2* [NAI+03]. The prototype transforms atomic services described in OWL-S into primitive HTN-tasks and composed services also described in OWL-S into non-primitive HTN-tasks. As user request an initial state and a partially ordered set of services has to be provided instead of the goal of the user. This means the user has to compose the needed services on a high level on his own. If the services in the user request are assembled out of other services then they are decomposed by the prototype into atomic services. This decomposition will be optimised according to a given optimisation rule in case several different decompositions are possible. This approach is more powerful than the Meteor-S prototype, since it can handle a replacement of two atomic activities by a superseding atomic activity. However like the Meteor-S approach the prototype presented by SIRIN ET AL. is very limited according it's composition power since it heavily relies on given human modelled service compositions. But this limitation in power has naturally a positive effect on composition complexity, making HTN-planning relatively fast in comparison to the forthcoming approaches.

Classical AI planning methods and especially their successors are a good fundament for full-automated service composition. Classical methods represent the planning domain as a deterministic state-transition system. In such approaches services are represented as transitions which change the state of the world. An initial state as well as the goal are represented by states within the state-transition system. Planning methods try to find a path of transitions—a service composition—from the initial state to the goal state [NM02, GNT04]. This proceeding reduces the planning problem to a search problem on states and transitions. MARTÍNEZ and LESPÉRANCE present in [ML04] a prototypical implementation of such a planner. The user can specify initial states and goals by using a logical language. After planning, the service composer returns a service composition which is capable of meeting the specified goal. This approach assume a deterministic behaviour of the services, which is a quite a limiting assumption for some real world domains. In [Mey04] MEYER presents an com-

position prototype, which indeed bases on a classical planning by state search. However the search algorithms has been extended in a way making it capable to deal with uncertain, non-deterministic results of services. The service composer is able to handle uncertainties by incorporating XOR-splits into the service composition. Further a graphical visualisation component for service compositions using graphical process representation is provided with the composer. PISTORE ET AL. present in [PBB+04] a different approach basing on planning as model checking. The main advantages of their approach are the consideration of non-determinism. This means the planner cannot foresee the actual interaction that will take place with external services and it cannot foresee the result of a service execution. However the planner relies on information about possible interactions and results of a service. Another interesting advantage are extended goals. Since model checking approaches work with temporal logic, it is possible to specify goal constraints which have to be met the whole time during the enactment of a service composition and not only at the end.

As recapitulation it has to be stated that current approaches towards full-automated service composition are in a immature state. While the fundamental algorithms are already known and well documented in AI books like [GNT04] there is a significant lack of matured application of these algorithms in the domain of services. Especially user-friendly methods for the specification of initial states, goals and service capabilities are missing. Furthermore the service composers are prototypical implementations not ready for a daily use. The reasons for this evaluation are missing robustness and composition power for real world scenarios as well performance issues due to missing optimisations towards the application domain.

Since full-automated service composition is essential for the exploration of the full potential of services further research on service composers is indispensable. A further area of application of full-automated service composers is presented in [SW03] and in more detail in [GMM+05]. It is proposed to use service composers not only for the creation of initial service compositions but also for exception handling by re-planning of service compositions which are in enactment. In case a service fails and this failure is not explicitly recovered by the service composition, the enactment of the composition has to be halted and the not-enacted part of the composition is re-planned in order to reach to user specified goal anyway. This is possible by retaining the goal and incorporating the results of all executed and failed services into the initial state of the user request. The application of such a strategy has the advantage of high self-adaptability according to changes in the web service world and that the service composer may create optimistic service compositions without trying to handle all possible failures in a composition. This make the compositions more concise and the planning less resource intensive.

4. Conclusion and Outlook

This survey presented an overview on the relationship of service composition and workflows and on methods for service composition ranging from a manual, semi- and full-automated composition to enable an evaluation. It can be concluded that tools for service composition are in a immature state and that there is still much research to do before service composition can be used easily and conveniently in real world scenarios. Reasons are the seriousness of the matter and the lack of widely used and elaborated formal and semantic service specifications and languages. Furthermore the term (automated) service composition is used differently by varying groups, making discussion and comparison not easy.

However the potential of automated services composition for the exploration of benefits of service oriented architectures is clear. For this reason further research in methods for service composition and re-composition, languages for semantic service specification and description of compositions is an imperative.

A. Service Composition Sample

This appendix denotes how a simplified service specification ontology, domain ontology and semantic service specifications may look like by using a coherent example. Furthermore the appearance of user requests is shown as well as the automated proceedings for composition and matchmaking of services are sketched. As formal, logical language the *F-Logic* [KLW95] dialect *Flora-2/XSB*[1] is used; for didactic reasons the usage of a simple and well known language for explanation of the sample scenario is more reasonable than the usage of the relatively new and still moving languages like OWL [MvH04] or WSML [dB05].

A.1. Service Specification Ontology

These service specification ontology describes concepts of the service domain. This are for example concepts like service, user requests and properties. In our example blow, the service specification ontology also includes descriptions on basic and enhanced data types like boolean or string and record or enumeration. Furthermore the ontology includes some validation rules to allow the syntactical validation of domain ontologies and service specifications.

```
// ****************************
// * Filename: asgOntology.flr *
// ****************************

// Validiation Constructs
// ================================================================================
// This ontology defines validation constructs to ensure valid specifications.
// Due to it is not possible to raise custom exceptions on insertion in Flora/XSB
// invalid specifications have to be queried manually after insertion by
// the following query (X is the object to validate, R is the reason for invalidity):
//
// X[], invalid(X, R).

// By default all objects are invalid if no valid statement is defined for them.
invalid(X, R) :- \+ valid(X), R = "1a not valid by default".

// The idea is that the "valid(X)" statement defines that an object X is valid
// in all cases for which no specific "invalid(X, R)" is defined.

// Classes
// ================================================================================
// Instances of class represent classes in the sense of the object oriented concept
class[].

// "class" itself is valid. Further, all subclasses if "class" are valid too.
valid(class).
valid(X) :- X::class.
```

[1] http://flora.sourceforge.net

```
// All subclasses of a class which is an instance of "class" are also instances of the
// concept "class":
Subclass:class    :- Subclass::Superclass, Superclass:class.

// Overloading of attributes while inheriting from a class is allowed, but the type of
// the attribute
// of the subclass has to be the same or a subtype to the type of the superclass.
invalid(X, R) :- X:class, X::SupX, X[A *=> T], SupX[A *=> SupT], \+ T = SupT,
                 \+ T::SupT, R = "1b type mismatch in attribute overloading".

// Unfortunately Flora/XSB shows all defines types for overloaded attributes.
// The following query finds the most restricted type of the overloaded attribute
// for a class named "xxx".
// _C = xxx, _C[A *=> T], lowestType(_C, A, T).
lowestType(C, A, T) :- C[A *=> T], C[A *=> SubT], \+ SubT = T, \+ SubT::T.
lowestType(C, A, T) :- C[A *=> T], \+ otherTypeExists(C, A, T).
otherTypeExists(C, A, T) :- C[A *=> T2], \+ T2 = T.

// All instances of class are invalid if they have an attribute which is not an instance of "class":
valid(X)      :- X:class.
invalid(X, R) :- X:class, X[_ *=> T], \+ T:class, \+ T::class,
                 R = "2a attribute type not instance or subclass of class".
invalid(X, R) :- X:class, X[_ => T], \+ T:class, \+ T::class,
                 R = "2b attribute type not instance or subclass of class".
invalid(X, R) :- X:class, X[_ *=>> T], \+ T:class, \+ T::class,
                 R = "2c attribute type not instance or subclass of class".
invalid(X, R) :- X:class, X[_ =>> T], \+ T:class, \+ T::class,
                 R = "2d attribute type not instance or subclass of class".

// All instances of classes are invalid if they have attribute values which have a type
// which is not matching the class definition.
valid(X)      :- X:C, C:class.
invalid(X, R) :- X:C, C:class, C[A *=> T], X[A -> V], \+ V:T,
                 R = "3a attribute value has wrong type".
invalid(X, R) :- X:C, C:class, C[A => T], X[A -> V], \+ V:T,
                 R = "3b attribute value has wrong type".
invalid(X, R) :- X:C, C:class, C[A *=>> T], X[A -> V], \+ V:T,
                 R = "3c attribute value has wrong type".
invalid(X, R) :- X:C, C:class, C[A =>> T], X[A -> V], \+ V:T,
                 R = "3d attribute value has wrong type".

// Hint: The current ontology does not enforce to set all attributes with values.
// So it is possible that some attribute values of class instances are not set.

// The "reification" is a special class which is used for the representation of
// logical expressions
// in F-Logic using the ${...} construct. These expressions are used for the
// specification of
// preconditions and effects of services.
reification:class.

// Relations
// ================================================================================
// Relations are a special concept used to specify a relation between two classes
relation[].

// Relations and all instances of them are valid.
valid(relation).
valid(X) :- X:relation.

// Grounded classes
// ================================================================================
// Grounded classes are classes which are representable in Java and XML, and which
// can be instantiated in Java and XML.
```

```
// Grounded classes are a special case of "class".
groundedClass::class[javaType *=> string,
                     xmlType  *=> string].

// "any" is a special class which is a superclass to all grounded classes.
// it should be only used within the ontology.
any:class.
Class::any :- Class:groundedClass.

// All subclasses of a class which is a grounded class are also grounded classes:
Subclass:groundedClass :- Subclass::Class, Class:groundedClass.

// These are the basic/atomic grounded types supperted by the ASG platform:
boolean:groundedClass[javaType -> "boolean":string,
                      xmlType  -> "xsd:boolean":string].
ordinal:groundedClass[javaType -> "long":string,
                      xmlType  -> "xsd:long":string].
float:groundedClass  [javaType -> "float":string,
                      xmlType  -> "xsd:float":string].
string:groundedClass [javaType -> "java.lang.String":string,
                      xmlType  -> "xsd:string":string].
binary:groundedClass [javaType -> "byte[]":string,
                      xmlType  -> "xsd:base64Binary":string].

// For convenience reasons Flora/XSB integers are mapped to ASG ordinals:
X:ordinal :- X:integer.

// A Record is a special case of a grounded class with an customized attribute/type definition.
// The types of the attributes have to be classes which are instances of groundedClass.
record::groundedClass.

// All records are represented in Java by "org.w3c.dom.Element" and in XML by an XML document fragment:
R[javaType -> "org.w3c.dom.Element":string]   :- R:record.
R[xmlType  -> "XML document fragment":string] :- R:record.

// All attribute types of a record have to be a grounded class.
invalid(X, R) :- X:record, X[_ => T], \+ T:groundedClass,
                 R = "6a attribute types must be member of groundedClass".
invalid(X, R) :- X:record, X[_ *=> T], \+ T:groundedClass,
                 R = "6b attribute types must be member of groundedClass".
invalid(X, R) :- X:record, X[_ =>> T], \+ T:groundedClass,
                 R = "6c attribute types must be member of groundedClass".
invalid(X, R) :- X:record, X[_ *=>> T], \+ T:groundedClass,
                 R = "6d attribute types must be member of groundedClass".

// Only single, inheriable attributes (*=>) are allowed for grounded classes!
invalid(X, R) :- X:record, X[_ =>> _], R = "6e =>> not allowed in record definitions".
invalid(X, R) :- X:record, X[_ *=>> _], R = "6f *=>> not allowed in record definitions".

// Restrictions are restricted types like enumerations or ranges which define restrictions on
// basic/atomic types/classes.
restriction::groundedClass[type   *=>  groundedClass].

// Enumerations are created by: _#:enumeration[type -> GROUNDEDCLASS, values ->> {V1, V2,...}].
enumeration::restriction[values *=>> any].
Enum[javaType -> JavaType] :- Enum[type -> Type], Type[javaType -> JavaType].
Enum[xmlType  -> XmlType]  :- Enum[type -> Type], Type[xmlType  -> XmlType].
Enum::Type                 :- Enum:enumeration, Enum[type -> Type].
V:Enum                     :- Enum:enumeration, Enum[values ->> V], V:T, Enum[type -> T].

// Some constraints on enumerations:
invalid(X, R) :- X:Enum, Enum:enumeration, \+ Enum[values ->> X],
                 R = "4a value not part of enumeration".
invalid(X, R) :- X:enumeration, X[type -> T], X[values ->> V], \+ V:T,
                 R = "4b enumeration type and value are not matching".
```

```
invalid(X, R) :- X:enumeration, \+ (X[type -> string]; X[type -> ordinal]; X[type -> float]),
                 R = "4c enums have to be string, ordinal or float".

// Ranges are created by: _#:range[type -> GROUNDEDCLASS, minimum -> V1, maximum -> V2].
range::restriction[minimum *=> any,
                   maximum *=> any].
Range[javaType -> JavaType] :- Range[type -> Type], Type[javaType -> JavaType].
Range[xmlType  -> XmlType]  :- Range[type -> Type], Type[xmlType  -> XmlType].
Range::Type                 :- Range:range, Range[type -> Type].
V:Range                     :- V:float, Range[minimum -> Min], Range[maximum -> Max],
                               V >= Min, V =< Max, V:T, Range[type -> T].
V:Range                     :- V:ordnial, Range[minimum -> Min], Range[maximum -> Max],
                               V >= Min, V =< Max, V:T, Range[type -> T].

// Some constraints on ranges:
invalid(X, R) :- X:range, X[type -> T], X[minimum -> V], \+ V:T,
                 R = "5a minimum has wrong type".
invalid(X, R) :- X:range, X[type -> T], X[maximum -> V], \+ V:T,
                 R = "5b maximum has wrong type".
invalid(X, R) :- X:range, \+ (X[type -> ordinal]; X[type -> float]),
                 R = "5c ranges have to be ordinal or float".
invalid(X, R) :- X:Range, Range:range, Range[minimum -> V], X < V,
                 R = "5d value smaller than range minimum".
invalid(X, R) :- X:Range, Range:range, Range[maximum -> V], X > V,
                 R = "5e value bigger than range maximum".

// Definition of service related concepts
// ================================================================================

// Parameters are special classes which are used to mark variables as input or output
// variables of a service. All variables which are marked as parameters in preconditions
// are input variables. All variables marked as parameters in positive effects are
// output variables of the service.
parameter:class.

// The hasValue relation is used to specify possible values of variables.
hasValue(_Variable, _Value):relation :- true.

// A Semantic Service Specification consists of at least one condition.
semanticServiceSpecification:class[conditions *=>> condition].

// Conditions consists of preconditions, (positive/negative) effects and queries.
// They are used to describe the possible
// different effects of a service according to a given precondition.
// Transformation of strings to reifications at run-time is not supported by Flora/XSB,
// so we have to store both variants of all
// precondition, positive/negative effects and queries.
// Conditions which are marked as exceptions are not used during normal planning, but
// they play a role in re-planning.
condition:class[precondR    *=> reification,
                precondS    *=> string,
                posEffR     *=> reification,
                posEffS     *=> string,
                posQueryR   *=> reification,
                posQueryS   *=> string,
                negEffR     *=> reification,
                negEffS     *=> string,
                negQueryR   *=> reification,
                negQueryS   *=> string,
                isException *=> boolean].

// Two conditions are equivalent if their reification and isException attributes are matching:
equivalentAttr(X, Y, A) :- X:condition, X[A -> P1], Y[A -> P2], P1 = P2.
equivalentAttr(X, Y, A) :- X:condition, \+ X[A -> _], \+ Y[A -> _].
```

```
equivalent(X, Y) :- equivalentAttr(X, Y, precondR),
                    equivalentAttr(X, Y, posEffR),
                    equivalentAttr(X, Y, posQueryR),
                    equivalentAttr(X, Y, negEffR),
                    equivalentAttr(X, Y, negQueryR),
                    equivalentAttr(X, Y, isException).

// oSP (=ordinalStringPair) is a tuple consisting of an ordinal and a string.
oSP:record[ord *=> ordinal,
           str *=> string].

// Service Grounding Specification holds all information which are needed to invoke
// the implementation of the service using C-5 interfaces
serviceGroundingSpecification:class[serviceImplRef *=>  string,
                                    operationName  *=>  string,
                                    inParamSeq     *=>> oSP,
                                    outParamSeq    *=>> oSP].


// Instances of provider represent the providers of services.
provider:record[name *=> string].

// provider names have to be unique
invalid(X, R) :- X:provider, X[name -> N], Y:provider, Y[name -> N], \+ X = Y,
                 R = "7a provider names have to be unique".

// Service property specifications types are special types of a record used for the specifications
// of static properties of services. Instances of those types are used to represent the results
// of negotiation, dynamic properties of services.
servicePropType::record.

// serviceProperties defines a minumum set of properties which have to be supported by a service.
serviceProperties:servicePropType[serviceName  *=> string,
                                  providerName *=> string].

// All subclasses of serviceProperties are also an instance of servicePropType.
P:servicePropType :- P::serviceProperties.

// Only subclasses of serviceProperties are valid values for servicePropType:
invalid(X, R) :- X:servicePropType, \+ X = serviceProperties, \+ X::serviceProperties,
                 R = "7b only subclasses of serviceProperties are valid values for servicePropType".

// Services consist of one or more specifications, an implementation reference
// and a property definition.
service:class[spec *=> semanticServiceSpecification].

// Composed Service
composedService::service[bpelRef *=> string].

// Atomic Service
atomicService::service[grounding   *=> serviceGroundingSpecification,
                       properties  *=> servicePropType].

// A user request consits of an initial state, a goal and a property
// definitions which is used as an constraint for planning an negotiation.
userRequest:class[initialStateR  *=>  reification,
                  initialStateS  *=>  string,
                  goalR          *=>  reification,
                  goalS          *=>  string,
                  propertyConstr *=>  servicePropType,
                  optCriteria    *=>> oSP].

// Query for executable services with new effects: (should be maybe extended
// by negative Effects in the future)
// Service[spec -> Spec], Spec[conditions ->> Cond], Cond[precondR -> Prec],
```

```
// Prec, Cond[posEffR -> _PosEff], \+ _PosEff.
```

A.2. Domain Ontology

The domain ontology defines the concepts used in a concrete application scenario. For example the concept of phone numbers in a the telecommunication scenario presented here.

```
// ********************************
// * Filename: domainOntology.flr *
// ********************************

// Include ASG-Ontology.
?- flAdd asgOntology.

// Definition of domain specific classes/records
// ============================================================================

// A phone provider is a special case of a provider.
phoneProvider::provider.

// A phone number is a special string.
phoneNumber::string.

// Coordinates used for location based services.
coordinate:record[longitude *=> float,
                  latitude  *=> float].

// A person has a name and a phone number:
person:record[name        *=> string,
              phoneNumber *=> phoneNumber].

// Maps are rectangular images of regions:
map:record[image      *=> binary,
           upperLeft  *=> coordinate,
           lowerRight *=> coordinate].

// Within our domain we support also cost and payment service properties.
// All service property definitions have inherit from minServProps.

minServProps::serviceProperties[cost    *=> float,
                                payment *=> paymentType:enumeration[type -> string,
                                  values ->> {"CreditCard":string, "EC":string, "none":string}]].

// For convenience reason we define some often used service property types.
zeroCostType:range[type -> float, minimum -> 0.0, maximum -> 0.0].
noPaymentType::paymentType[type -> string, values ->> {"none":string}].

// Definition of domain specific relationships and rules
// ============================================================================
// Maps may have a cross at a specific coordinate:
hasCross(map, coordinate):relation.

// Some phone numbers habe a known provider.
providerOf(phoneNumber, phoneProvider):relation.

// phone numbers and persons may be located at a specific coordinate.
coordinateOf(phoneNumber, coordinate):relation.
coordinateOf(person, coordinate):relation.

// If the coordinate of a phone number is known, then it is assumed that the owner
// of the phone number is located at the same coordinates.
coordinateOf(X, C) :- X[phoneNumber -> P], P:phoneNumber, coordinateOf(P, C).
```

```
// Definition of domain specific objects/instances
// ================================================================================
telco:phoneProvider[name -> "Telco":string].
vfone:phoneProvider[name -> "Vfone":string].
hpi:provider[name -> "HPI":string].
falk:provider[name -> "Falk":string].
```

A.3. Service Specifications

The following code sniplets specify the semantics of various services of our scenario.

```
fpn:atomicService[
  spec    -> fpnSpec:semanticServiceSpecification[
               conditions ->> fpnCond:condition[
               precondR -> ${N:string, N:parameter, X:person, X[name->N]}:reification,
               precondS ->  "N:string, N:parameter, X:person, X[name->N]":string,
               posEffR  -> ${P:phoneNumber, P:parameter, X[phoneNumber->P]}:reification,
               posEffS  ->  "P:phoneNumber, P:parameter, X[phoneNumber->P]":string]],
  grounding -> fpnBridge:serviceGroundingSpecification[
                  serviceImplRef -> "fpnRef":string,
                  operationName  -> "doIt":string,
                  inParamSeq     ->> {_#:oSP[ord -> 1, str -> "N":string]},
                  outParamSeq    ->> {_#:oSP[ord -> 1, str -> "P":string]}],
  properties -> fpnProps::minServProps[
                  serviceName  *=> fpnSNType:enumeration[type -> string,
                                                         values ->> {"findPhoneNumber":string}],
                  providerName *=> fpnPNType:enumeration[type -> string,
                                                         values ->> {"HPI":string}],
                  cost         *=> zeroCostType,
                  payment      *=> noPaymentType]].
```

This service describes the *findPhoneNumber* service, which is provided by the Provider *HPI*. This service takes a string, which is assigned to the variable *N*, as input parameter and it delivers a *phoneNumber* as output parameter, which is assigned to the variable *P*. For the specification of preconditions and effects we need further the variable *X* of type *person*. This variable no direct relation with the service implementation, it is just needed for the a formal semantic specification. The precondition of the service is fulfilled if the input parameter *N* is a name of a *person* (`X[name->N]`). The effect of the service execution is, that a *phoneNumber* is returned by the service, which belongs to the person *X* (`X[phoneNumber->P]`).

Hint: The service object defined above is identified by *fpn*. The identification string is not of interest, because the real name of the service is stored in the static property name of the service. In an full-fledged implementation the object identification strings should be generated automatically.

```
fpp:atomicService[
  spec    -> fppSpec:semanticServiceSpecification[
               conditions ->> fppCond:condition[
               precondR  -> ${P:phoneNumber, P:parameter}:reification,
               precondS  ->  "P:phoneNumber, P:parameter":string,
               posEffR   -> ${N:string, N:parameter, QueryX:phoneProvider[name->N],
                              providerOf(QueryX, P)}:reification,
               posEffS   ->  "N:string, N:parameter, QueryX:phoneProvider[name->N],
                              providerOf(QueryX, P)":string,
               posQueryR -> ${QueryX:phoneProvider}:reification,
               posQueryS ->  "QueryX:phoneProvider":string]],
  grounding -> fpnBridge:serviceGroundingSpecification[
```

```
              serviceImplRef -> "telekomRef":string,
              operationName  -> "fpp":string,
              inParamSeq     ->> {_#:oSP[ord -> 1, str -> "P":string]},
              outParamSeq    ->> {_#:oSP[ord -> 1, str -> "N":string]}],
  properties -> fppProps::minServProps[
                serviceName  *=> fppSNType:enumeration[type -> string,
                                                       values ->> {"findPhoneProvider":string}],
                providerName *=> fppPNType:enumeration[type -> string,
                                                       values ->> {"Telco":string}],
                cost         *=> fppCostType:range[type -> float, minimum -> 1.0, maximum -> 3.0],
                payment      *=> fppPayType::paymentType[type -> string,
                                                         values ->> {"CreditCard":string}]]].

gpn:atomicService[
  spec    -> gpnSpec:semanticServiceSpecification[
              conditions ->> gpnCond:condition[
              precondR  -> ${P:phoneNumber, P:parameter}:reification,
              precondS  ->  "P:phoneNumber, P:parameter":string,
              posEffR   -> ${N:string, N:parameter, QueryX:phoneProvider[name->N],
                             providerOf(QueryX, P)}:reification,
              posEffS   ->  "N:string, N:parameter, QueryX:phoneProvider[name->N],
                             providerOf(QueryX, P)":string,
              posQueryR -> ${QueryX:phoneProvider}:reification,
              posQueryS ->  "QueryX:phoneProvider":string]],
  grounding -> fpnBridge:serviceGroundingSpecification[
                serviceImplRef -> "vodafoneRef":string,
                operationName  -> "gpn":string,
                inParamSeq     ->> {_#:oSP[ord -> 1, str -> "P":string]},
                outParamSeq    ->> {_#:oSP[ord -> 1, str -> "N":string]}],
  properties -> gpnProps::minServProps[
                 serviceName  *=> gpnSNType:enumeration[type -> string,
                                                        values ->> {"getProviderName":string}],
                 providerName *=> gpnPNType:enumeration[type -> string, values ->> {"Vfone":string}],
                 cost         *=> gpnCostType:range[type -> float, minimum -> 2.0, maximum -> 4.0],
                 payment      *=> gpnPayType::paymentType[type -> string, values ->> {"EC":string}]]].
```

The precondition of both services is that the required input parameter (phone umber *P*) is delivered. The costs of the service execution is not fixed for boths services (for example it may depends on the load). It has to be negotiated at runtime. Nevertheless static properties for costs are given to define the range of possible values. In case of the *findPhoneProvider* service the costs are known to be between 1 and 3 and in case of the *getProviderName* service the costs are between 2 and 4.

Having a closer look at both services we see, that both services are semantically identical. They have equivalent service specifications. This means: both services have the same input, output, preconditions and effects. This phenomenon is called synonymy. Usually most programmers and system designers try to avoid such a situation by giving semantically identical services the same name. But practical experience shows, that this proceeding can not be keept up if a larger amount of partners are involved into a project for a longer time period. For this reason a semantic service platform should be able to deal with such situations.

```
gpl1:atomicService[
  spec    -> gpl1Spec:semanticServiceSpecification[
              conditions ->> gpl1Cond:condition[
              precondR -> ${P:phoneNumber, P:parameter,  providerOf(Q, P), Q[name -> "Telco"]}:reification,
              precondS ->  "P:phoneNumber, P:parameter,  providerOf(Q, P),  Q[name -> ""Telco""]":string,
              posEffR  -> ${C:coordinate, C:parameter, coordinateOf(P, C)}:reification,
              posEffS  ->  "C:coordinate, C:parameter, coordinateOf(P, C)":string]],
  grounding -> fpnBridge:serviceGroundingSpecification[
                serviceImplRef -> "telekomRef":string,
```

```
                operationName  -> "gpl":string,
                inParamSeq     ->> {_#:oSP[ord -> 1, str -> "P":string]},
                outParamSeq    ->> {_#:oSP[ord -> 1, str -> "C":string]}],
  properties -> gpl1Props::minServProps[
                 serviceName  *=> gpl1SNType:enumeration[type -> string,
                                                        values ->> {"getPhoneLocation":string}],
                 providerName *=> gpl1PNType:enumeration[type -> string,
                                                        values ->> {"Telco":string}],
                 cost         *=> gpl1CostType:range[type -> float, minimum -> 10.0, maximum -> 10.0],
                 payment      *=> gpl1PayType::paymentType[type -> string,
                                                          values ->> {"CreditCard":string}]]].

gpl2:atomicService[
  spec    -> gpl2Spec:semanticServiceSpecification[
              conditions ->> gpl2Cond:condition[
              precondR -> ${P:phoneNumber, P:parameter,  providerOf(Q, P), Q[name -> "Vfone"]}:reification,
              precondS ->  "P:phoneNumber, P:parameter,  providerOf(Q, P),  Q[name -> ""Vfone""]":string,
              posEffR  -> ${C:coordinate, C:parameter, coordinateOf(P, C)}:reification,
              posEffS  ->  "C:coordinate, C:parameter, coordinateOf(P, C)":string]],
  grounding -> fpnBridge:serviceGroundingSpecification[
                 serviceImplRef -> "vodafoneRef":string,
                 operationName  -> "gpl":string,
                 inParamSeq     ->> {_#:oSP[ord -> 1, str -> "P":string]},
                 outParamSeq    ->> {_#:oSP[ord -> 1, str -> "C":string]}],
  properties -> gpl2Props::minServProps[
                 serviceName  *=> gpl2SNType:enumeration[type -> string,
                                                        values ->> {"getPhoneLocation":string}],
                 providerName *=> gpl2PNType:enumeration[type -> string,
                                                        values ->> {"Vfone":string}],
                 cost         *=> gpl2CostType:range[type -> float, minimum -> 10.0, maximum -> 10.0],
                 payment      *=> gpl2PayType::paymentType[type -> string, values ->> {"EC":string}]]].
```

The two services above have a slightly different semantic (the first one works only for phone numbers provided by *Telco* and the second one only for phone numbers provided by *Vfone*). Both services have been assigned the same name. This phenomenon is called homonymy (or to be more exact homography). Like synonymy this phenomenon appears when serveral people from different areas are working together on a huge system. For this reason our ASG-system should be able to deal with such situations. The costs of service execution is for both services exactly 10, independently from any run-time aspects.

```
gm:atomicService[
  spec    -> gmSpec:semanticServiceSpecification[
               conditions ->> gmCond:condition[
               precondR -> ${C:coordinate, C:parameter}:reification,
               precondS ->  "C:coordinate, C:parameter":string,
               posEffR  -> ${M:map, M:parameter}:reification,
               posEffS  ->  "M:map, M:parameter":string]],
  grounding -> fpnBridge:serviceGroundingSpecification[
                 serviceImplRef -> "falksRef":string,
                 operationName  -> "getMap":string,
                 inParamSeq     ->> {_#:oSP[ord -> 1, str -> "C":string]},
                 outParamSeq    ->> {_#:oSP[ord -> 1, str -> "M":string]}],
  properties -> gmProps::minServProps[
                 serviceName  *=> gmSNType:enumeration[type -> string, values ->> {"getMap":string}],
                 providerName *=> gmPNType:enumeration[type -> string, values ->> {"Falk":string}],
                 cost         *=> gmCostType:range[type -> float, minimum -> 15.0, maximum -> 15.0],
                 payment      *=> gmPayType::paymentType[type -> string,
                                   values ->> {"CreditCard":string, "EC":string}]]].

mc:atomicService[
  spec    -> mcSpec:semanticServiceSpecification[
```

```
            conditions ->> mcCond:condition[
            precondR -> ${M1:map, M1:parameter, C:coordinate, C:parameter}:reification,
            precondS ->  "M1:map, M1:parameter, C:coordinate, C:parameter":string,
            posEffR  -> ${M2:map, M2:parameter, hasCross(M2, C)}:reification,
            posEffS  ->  "M2:map, M2:parameter, hasCross(M2, C)":string]],
  grounding -> fpnBridge:serviceGroundingSpecification[
              serviceImplRef -> "falksRef":string,
              operationName  -> "makeCross":string,
              inParamSeq     ->> {_#:oSP[ord -> 1, str -> "M1":string],
                                  _#:oSP[ord -> 2, str -> "C":string]},
              outParamSeq    ->> {_#:oSP[ord -> 1, str -> "M2":string]}],
  properties -> gmProps::minServProps[serviceName  *=> gmSNType:enumeration[
               type -> string, values ->> {"makeCross":string}],
               providerName *=> gmPNType:enumeration[
                                type -> string, values ->> {"Falk":string}],
                                cost         *=> zeroCostType,
                                payment      *=> noPaymentType]].
```

The last two definitions specify a service providing a map and a service which is able to make a cross on a map at an arbitrary position.

A.4. User Request

In this scenario we presume that a user wants to find his buddy, and that there is no composed service stored in the system to solve this problem. To allow an automated service composition, the request of the user has to be specified in a formal manner:

```
ur:userRequest[initialStateR -> {N:string, N:parameter, hasValue(N, "Dominik Kuropka"),
                                 X:person, X[name->N]}:reification,
               initialStateS -> "N:string, N:parameter, hasValue(N, ""Dominik Kuropka""),
                                 X:person, X[name->N]":string,
               goalR         -> {M:map, M:parameter, C:coordinate, coordinateOf(X, C),
                                 hasCross(M, C)}:reification,
               goalS         -> "M:map, M:parameter, C:coordinate, coordinateOf(X, C),
                                 hasCross(M, C)":string,
               propertyContr -> minServProps,
               optCriteria   -> _#:oSP[ord -> 1, str -> "min(cost)"]].
```

This formal specification is equal to the following natural language problem specification: "I (the user) know a name *N* of a person. The name is "Dominik Kuropka". This is the initial state of the problem. My problem is solved if I have reached a state, in which the coordinates *C* of the person *N* are known and I get a map *M* having a cross at those coordinates *C*." Further the selected services have to ensure the property constraint. Because minServProps is the most general class of properties, in fact no constraint is specified in this case.

Before service composition can start the ontologies and service specifications have to be loaded and the initial state has to be transformed into a valid Flora/XSB insert statement. Besides wrapping the initial state inside an insert statement the variables like the name *N* have to be replaced by objects like *urN* (stands for variable *N* of the user request). The initial state has to be stored in the Flora reasoning engine by the use of the following commands:

```
flora2 ?- [serviceSpecifications].
flora2 ?- insert{urN:string, urN:parameter, hasValue(urN, "Dominik Kuropka"),
                 urX:person, urX[name->urN]}.
```

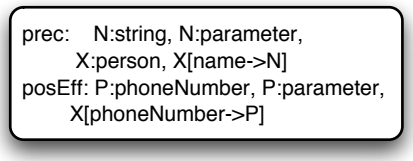


Figure 5.: Composition after first step.

A.5. Sketch of full-automated Service Composition

The composition is stated by querying the executable services in a given state. The generic Flora query and the given result is presented next:

```
flora2 ?- Service[spec -> Spec], Spec[conditions ->> Cond], Cond[precondR -> Prec],
          Prec, Cond[posEffR -> _PosEff], \+ _PosEff.

Service = fpn
Prec = (${urN:string}, (${urN:parameter}, (${urX:person}, ${urX[name -> urN]})))
```

The above query states, that we are looking for a service with a service specification, that has preconditions which are fulfilled and effects which are not fulfilled in the actual state. The result for the initial state is only the *findPhoneNumber* service. Further the result tells us which variables (in this case *urN*) are assigned to the parameters. The next step is the virtual execution of the found service. This means the state has to be adjusted according to the service's effects:

```
flora2 ?- insert{fpnP:phoneNumber, fpnP:parameter, urX[phoneNumber->fpnP]}.
```

As for the initial state the variables inside the service's effects must be replaced by concrete objects (sample: *fpnP* stands for the variable *P* of the service with the object identifier *fpn*). Therefore a new *phoneNumber* object *fpnP* is added to the reasoners database. The *phoneNumber* object is assigned to the person as its phone number. Figure 5 shows the state of the composition after this first step.

Again the executable services must be found. The already presented query for this task will be used again, resulting in the following list of executable services:

```
flora2 ?- Service[spec -> Spec], Spec[conditions ->> Cond],
          Cond[precondR -> Prec], Prec, Cond[posEffR -> _PosEff], \+ _PosEff.

Service = fpp
Prec = (${fpnP:phoneNumber}, ${fpnP:parameter})

Service = gpn
Prec = (${fpnP:phoneNumber}, ${fpnP:parameter})
```

As stated in the service specification both services are synonymous. This can be shown using the `equivalent(X,Y)` rule. It returns *yes* if the service specifications *X* and *Y* are equivalent (the service specifications have to have equivalent preconditions and effects):

```
flora2 ?- fpp[spec ->> S1], gpn[spec ->> S2], equivalent(S1,S2).

Yes
```

Because both services are equivalent, the decision which service is used, should be done during runtime through negotiation. For this reason the services are merged in the composed service. As it is

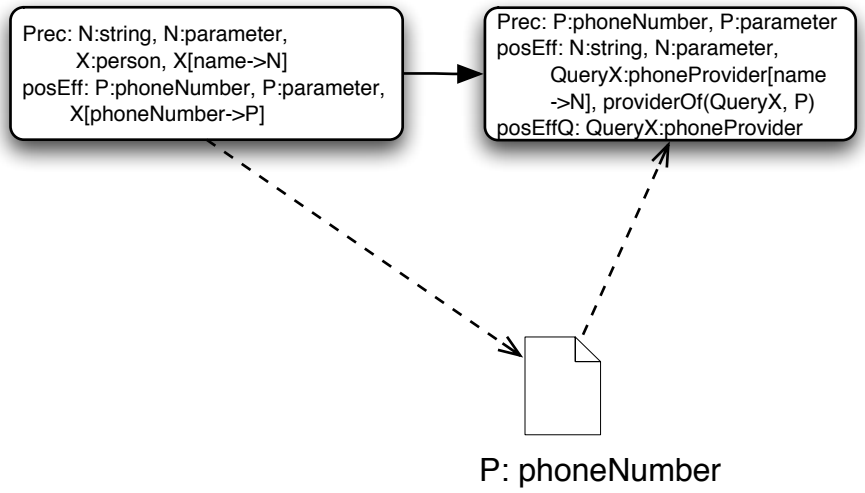


Figure 6.: Composition after second step.

shown in figure 6, a service specification is inserted into the service composition that is equivalent to both service specifications.

The merged service has to be be virtually executed:

```
insert{fppgpnN:string, fppgpnN:parameter, QueryX:phoneProvider[name->fppgpnN],
       providerOf(QueryX, fpnP) | QueryX:phoneProvider}.
```

The service has more than one possible effects. The provider could be either *Telco* or *Vfone*. Actually the above query is not correct regarding all details. It inserts both possible effects into the state. To work correctly the planer has to work with two different states from now on. This is not displayed here because the result of both approaches are the same in this special case. Again the executable services for the (two merged) states have to be queried:

```
flora2 ?- Service[spec -> Spec], Spec[conditions ->> Cond],
          Cond[precondR -> Prec], Prec, Cond[posEffR -> _PosEff], \+ _PosEff.

Service = gpl1
Prec = (${fpnP:phoneNumber}, (${fpnP:parameter}, (${providerOf(telco, fpnP)},
        ${telco[name -> "Telco"]})))

Service = gpl2
Prec = (${fpnP:phoneNumber}, (${fpnP:parameter}, (${providerOf(vfone, fpnP)},
        ${vfone[name -> "Vfone"]})))
```

One service is executable in each of the possible states. If the provider of the phone numer is *Telco* then the service *gpl1* is executable. If the provider is *vfone* the service *gpl2* is executable. Because the location has to be found, one of the services has to be executed alternatively. We have to add the effects of both services. This is unproblematic here as both services have the same effect. For this reason only one insert statement is necessary:

```
flora2 ?- insert{gplC:coordinate, gplC:parameter, coordinateOf(fpnP, gplC)}.
```

Because two services have been selected, both must be added to the composition as shown in figure 7.

This effects that a new *coordinate* object exists and this new *coordinate* object is the coordinate of the given *phoneNumber*. According to the world rules a person has the same location as its phone. Now the location of the person and its phone are known and we have to request a map. Lets see if the reasoner thinks the same:

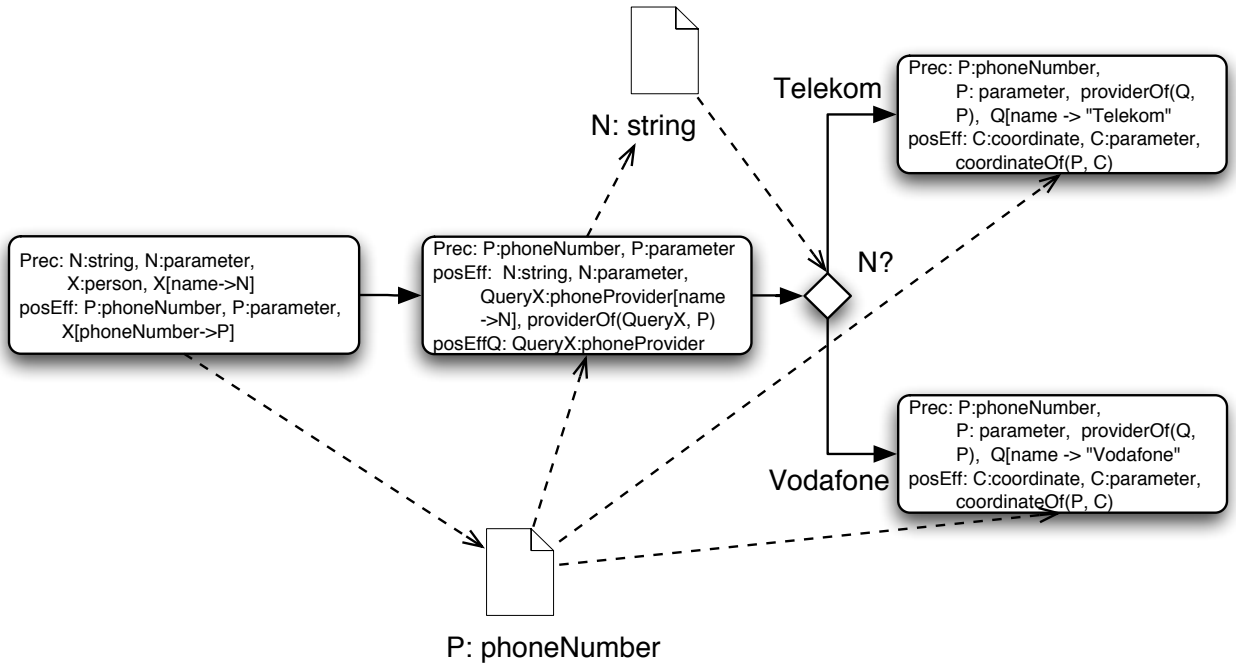


Figure 7.: Composition after third step.

```
flora2 ?- Service[spec -> Spec], Spec[conditions ->> Cond],
          Cond[precondR -> Prec], Prec, Cond[posEffR -> _PosEff], \+ _PosEff.

Service = gm
Prec = (${gplC:coordinate}, ${gplC:parameter})
```

Only the *getMap* service is exectuable at the moment. Therefore it is selected and virtually executed:

```
insert{gmM:map, gmM:parameter}.
```

A new *map* oject is created. The two alternative execution flows for the different location services are merged as shown in figure 8.

Finally the position in the map has to be marked by a cross. The query retrieves the following executable services:

```
flora2 ?- Service[spec -> Spec], Spec[conditions ->> Cond],
          Cond[precondR -> Prec], Prec, Cond[posEffR -> _PosEff], \+ _PosEff.

Service = mc
Prec = (${gmM:map}, ${gmM:parameter}, (${gplC:coordinate}, ${gplC:parameter}))
```

Only the *makeCross* service executable. Its virtual execution looks like this:

```
insert{mcM2:map, mcM2:parameter, hasCross(mcM2, gplC)}.
```

The composition including also the last selected service is shown in figure 9. We can test if we can execute some more services and if the user specified goal is reached:

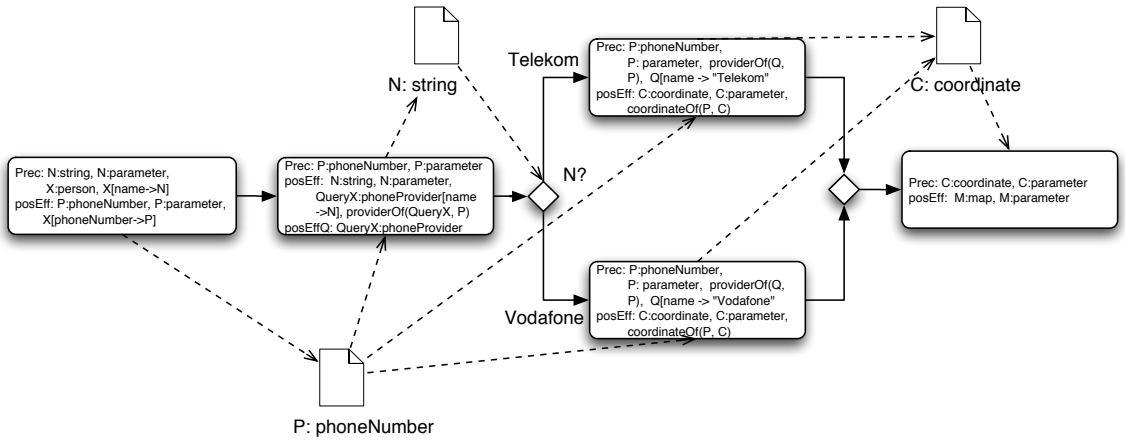


Figure 8.: Composition after fourth step.

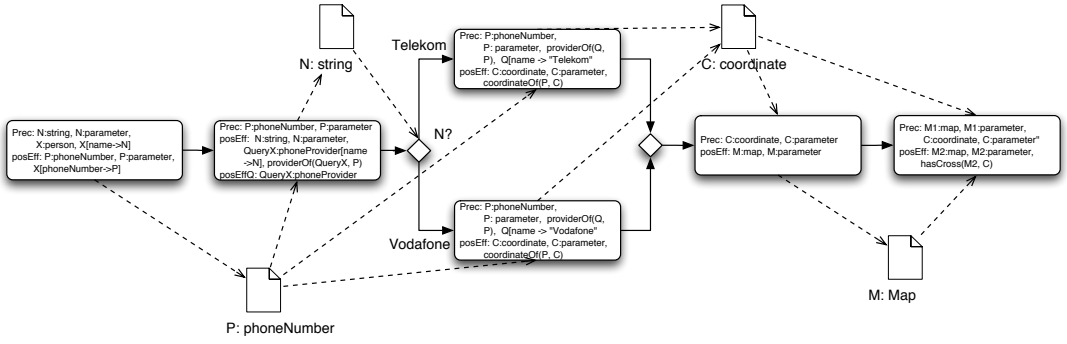


Figure 9.: Composition after fifth and final step.

```
flora2 ?- Service[spec -> Spec], Spec[conditions ->> Cond],
          Cond[precondR -> Prec], Prec, Cond[posEffR -> _PosEff], \+ _PosEff.

No

flora2 ?- M:map, M:parameter, C:coordinate, coordinateOf(X, C), hasCross(M, C).

Yes
```

Note: In this simple example the planner did not need to do any service selections. In a more complex example possibly several service with different effects are executable in a given state. The planner has to select the service that makes the most sense. Further optimisation criteria (like: I want it as cheap or fast as possible) and static restrictions (like: I want only that services of the provider X and Y are used) are not taken into account in this scenario.

Bibliography

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, and Johannes Klein et. al. Business process execution language for web services version 1.1. Technical report, BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, 2003.

[AVMM04] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint driven web service composition in meteor-s. In *Proceedings of IEEE SCC 2004*, 2004.

[BPN04] Business Process Management Initiative. *Business Process Modeling Notation (BPMN) Version 1.0*, 2004.

[Bur00] Steve Burbeck. The tao of e-business services — the evolution of web applications into service-oriented components with web services. Technical report, IBM, 2000.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addision Wesley Publishing Company, 1999.

[CS02] Jorge Cardoso and Amit Sheth. Semantic e-workflow composition. Technical report, LSDIS Lab, Department of Computer Science at the University of Georgia, 2002.

[dB05] Jos de Bruijn, editor. *The Web Service Modeling Language WSML*, 2005.

[GMM+05] Michal Gajewski, Harald Meyer, Mariusz Momotko, Hilmar Schuschel, and Mathias Weske. Dynamic failure recovery of generated workflows. In *Proceedings of the 16th International Conference and Workshop on Database and Expert Systems Applications*, pages 982 – 986. IEEE Computer Society Press, 2005. 1st workshop on Business Process Monitoring and Performance Management.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Morgan Kaufmann, 2004.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, May 1995.

[Kur04] Dominik Kuropka. *Modelle zur Repräsentation natürlichsprachlicher Dokumente – Information-Filtering und -Retrieval mit relationalen Datenbanken*. Logos Verlag, Berlin, 2004.

[Mar04] David Martin, editor. *OWL-S: Semantic Markup for Web Services*, 2004.

[Mey04] Harald Meyer. Entwicklung und realisierung einer planungskomponente für die komposition von diensten. Master's thesis, University of Potsdam, 2004.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[ML04] Erick Martínez and Yves Lespérance. Web service composition as a planning task: Experiments using knowledge-based planning. In *Workshop on Planning and Scheduling for Web and Grid Services in conjunction with ICAPS 2004*, Whistler, Canada, 2004.

[MvH04] Deborah L. McGuinness and Frank van Harmelen, editors. *OWL Web Ontology Language Overview*. Web Ontology Working Group at the World Wide Web Consortium (W3C), 2004.

[NAI+03] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Okhtay Ilghami, J. William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[NM02] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, pages 77–88, 2002.

[OAS02] Organization for the Advancement of Structured Information Standards (OASIS). *UDDI Version 2 Specifications*, 2002.

[PBB+04] Marco Pistore, F. Barbon, Piergiorgio Bertoli, D. Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *Workshop on Planning and Scheduling for Web and Grid Services in conjunction with ICAPS 2004*, Whistler, Canada, 2004.

[Pet66] Carl Adam Petri. Kommunikation mit automaten. *Schriften des IIM*, 2, 1966.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[RLK05] Dumitru Roman, Holger Lausen, and Uwe Keller, editors. *Web Service Modeling Ontology (WSMO)*, 2005.

[RRD04] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Correctness criteria for dynamic changes in workflow systems——a survey. *Data & Knowledge Engineering*, 50(1):9–34, 2004.

[SAP97] SAP AG, Walldorf, Germany. *WF SAP Business Workflow*, 1997.

[Sch00] August-Wilhelm Scheer. *ARIS - Business Process Modeling*. Springer, 3 edition, 2000.

[SHP02] 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 conjunction with ICEIS 2003*, 2002.

[Smi02] Barry Smith. Ontology and information systems, 2002.

[Sof99] Software-Ley GmbH, Pullheim, Germany. *COSA 3.0 User Manual*, 1999.

[SPW+04] Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau. Htn planning for web service composition using shop2. *Journal of Web Semantics*, 1(4):377–396, 2004.

[Sta00] Staffware plc, Berkshire, United Kingdom. *Staffware 2000 / GWD User Manual*, 2000.

[SW03] Hilmar Schuschel and Mathias Weske. Integrated workflow planning and coordination. In *14th International Conference on Database and Expert Systems Applications*, pages 771–781, 2003.

[vdAtHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.

[W3C01] World Wide Web Consortium. *Web Services Description Language (WSDL) 1.1*, 2001.

ISBN 3-937786-78-3
ISSN 1613-5652