Master Thesis

Supporting the Modeling of Business Processes Using Semi-Automated Web Service Composition Techniques

Jan Schaffner

Supervisors
Prof. Dr. Mathias Weske, Hasso-Plattner-Institute, Potsdam, Germany
Dipl.-Inform. Cafer Tosun, SAP Labs, Palo Alto, USA
Dipl.-Inform. Harald Meyer, Hasso-Plattner-Institute, Potsdam, Germany

December 22, 2006

Abstract

When creating service compositions from a very large number of atomic service operations, it is inherently difficult for modelers to discover suitable operations for their specific goals. Automated service composition claims to solve this problem, yet, it will only work when complete and correct ontologies along with service descriptions are in place.

In this thesis, we present a semi-automated modeling environment for Web service compositions. At every step in the process of creating the composition, the environment suggests a number of relevant Web services the modeler. Furthermore, the environment summarizes the problems that would prevent the composed service from being invocable. The environment is also able to insert composed services into the composition at suitable places, so that the atomic services of the inserted composition produce data artifacts that are missing in the parent service composition.

The main contributions of this thesis are the definition of three mixed initiative features for semi-automated composition based on a rigorous formal model, an industry scenario underpinning the usefulness of these features, and a prototypical implementation demonstrating their applicability.

Our results show that this mixed initiative approach significantly alleviates the creation of composed services. We validated our implementation with the help of SAP, the leading vendor of business applications, using their processes and their service repository, which spans across multiple functional areas of enterprise computing.

Zusammenfassung

Bei der Dienstkomposition mit einer großen Anzahl von Basisoperationen fällt es dem Modellierer häufig schwer, die passenden Operationen für sein konkretes Ziel auszuwählen. Die automatische Dienstkomposition stellt eine Lösung für dieses Problem dar, jedoch funktioniert diese ausschließlich, wenn vollständiges und korrekt spezifiziertes Domänenwissen in Form von Ontologien sowie semantische Dienstbeschreibungen verfügbar sind.

In dieser Arbeit wird eine Modellierungsumgebung für die semi-automatische Komposition von Web Services vorgestellt. Diese Umgebung schlägt dem Modellierer in jedem Schritt der Erstellung der Dienstkomposition relevante Web Services vor. Weiterhin werden die Probleme zusammengefasst, welche die Ausführung der zu erstellenden Dienstkomposition verhindern würden. Der Editor ist außerdem in der Lage, Teilkompositionen an passenden Stellen in eine Dienstkomposition derart einzufügen, dass die Operationen der Teilkomposition fehlende Datenartefakte der übergeordneten Komposition bereitstellen.

Die wesentlichen Beiträge dieser Arbeit sind die Definition der drei "mixed initiative" Eigenschaften von semi-automatischer Dienstkomposition auf der Basis eines formalen Modells. Zudem wird die Nützlichkeit dieser Eigenschaften durch ein Anwendungsszenario aus der Industrie belegt. Die praktische Anwendbarkeit des vorgestellten Ansatzes wird durch eine prototypische Implementierung einer Modellierungsumgebung gezeigt, welche die "mixed initiative" Eigenschaften unterstützt.

Die Ergebnisse der Arbeit zeigen, dass der vorgestellte Ansatz die manuelle Erstellung von Dienstkompositionen deutlich vereinfacht. Die Implementierung wurde zusammen mit SAP, dem führenden Anbieter betriebswirtschaftlicher Anwendungen, validiert. Als Ausgangsbasis für die Validierung wurden die Geschäftsprozesse und das Service Repository von SAP herangezogen, das sich über verschiedenste funktionale Bereiche betriebswirtschaftlicher Software erstreckt.

Acknowledgements

I would like to thank Prof. Dr. Mathias Weske for supervising this work. I would also like to thank Harald Meyer for the many insightful conversations and hints during the course of writing this thesis.

I also wish to thank Dr. Vishal Sikka for the opportunity to carry out the necessary research in his group at SAP Labs in Palo Alto. In particular, I would like to thank Cafer Tosun for his ongoing support of this work. Furthermore, I would like to thank Heinz Roggenkemper, Natalia Shmoilova, Shuyuan Chen and Prof. Dr. Hasso Plattner for inspiring discussions.

I would also like to thank my parents for their support and encouragement and Julia for her enormous understanding and patience.

Contents

List of Figures

List of Tables

Listings

1. Introduction

Industry has recognized business process management systems (BPMS) [68] as a way to realize their strategic focus on business processes [46]. At the same time, Web services have been evolving as a promising approach to provide value added functionality across organizational borders. They can greatly ease the integration of distributed software systems. Standards like XLANG [64] or the Web Services Business Process Execution Language (WS-BPEL [19]) allow using Web services as a means to realize business processes. In doing so, the actual Web service operations represent activities in business processes. Due to the technologies underlying Web services ([25, 49, 11]), business processes that are composed of Web services can involve multiple business parties in a transparent manner. According to Forrester Research [23], many currently available BPMSs are service-oriented and offer one way to implement composite applications in service-oriented environments. Business processes can be modeled as compositions of Web service operations. In general, these compositions are created manually: A process expert creates a static process model which can be translated into an executable language, i.e. WS-BPEL, and, then, be enacted. Yet, the manual creation of Web service compositions presents a very complex task, which is due to the following problems:

At design time, modelers have to anticipate all possible cases that shall be handled by the process they are creating: All imaginable alternative paths in the execution of the process flow have be thought of and specified. Also, all failures that could possibly arise during execution must be considered. The modelers have to interpret the names, interfaces and - if applicable - textual descriptions of the services they use in the composition in order to understand their capabilities and nonfunctional properties. That is a prerequisite for the delicate task of correctly defining both the control and data flow among the service operations. Due to this complexity, it seems likely that the modeler introduces errors into the service composition. Handcrafted service compositions are rarely optimal as they contain tradeoffs. They are likely to become complex and hard to change, and prove, thus, difficult and expensive to maintain.

In recent years, the above mentioned reasons have served as a rationale to automate the creation of the Web service compositions ([71, 51, 60, 8]). Academia has proposed systems that automatically create executable plans for each individual case at runtime. This opposes the idea of creating composed services that cover as many cases as possible. The plans are produced in a fully automated fashion, based on domain knowledge, i.e. ontologies [24], and semantic service descriptions.

While automated planners are able to reduce complexity, inflexibility and error-proneness akin to the creation of composed services, several drawbacks can be identified: Automated planning relies on the availability of complete formal representations of the domain knowledge and the individual cases that need to be resolved, i.e. their initial state and goal state need to be formally encoded.

The task of formally specifying a domain in sufficient fidelity, so that it can be used for automated planning presents a huge challenge. Especially for complex domains we can legitimately assume that complete ontologies will not be available in the near future. Incomplete domain knowledge, however, will often result in a situation that an automated planner fails to produce a plan. Erroneous domain knowledge, moreover, can result in situations where a planner finds wrong plans. In contrast, human planners can draw upon their experience with a specific domain when they create a composed service. This experience will often compensate for missing or erroneous ontologies.

Moreover, the fact that fully automated service composition methods do not require a human being in the loop poses an organizational and juridical impediment: In business reality it is required that concrete persons are responsible for a particular business process. This has lowered industry acceptance of automated planning techniques, thus, slowing down the transition from research to industry.

The goal of this thesis is to show that the techniques of automated planning can be used to alleviate the manual creation of business processes by a human planner. The incorporation of the matchmaking technologies as used by automated planners into a semi-automated modeling tool for creating enterprise service compositions has several advantages: On the one hand, the problems of complexity, inflexibility and error-proneness akin to service composition can be reduced or even eliminated by the aid of new ‘mixed initiative features’, which can be built on top of current Semantic Web [9] technologies. On the other hand, modelers can rely on their experience in the creation of business processes. They can use that knowledge to compensate for the lack of fully-fledged ontologies. This can help solve the problem of planning with incomplete information faced by fully automated planning environments.

1.1 Preliminary Definitions

The purpose of this section is to clarify basic concepts that will frequently be referred to throughout this thesis.

1.1.1 Service

The term ‘service’ is often used with various meanings. In general, a service is referred to as an abstract set of functionality. Laures [38] introduces a technology-

oriented service layer model to classify the various ambiguous meanings of this term. The term 'service' as used in this thesis resides on the Web services layer of this model. By service we understand a software component that exposes different operations via an interface specified in the Web Services Description Language (WSDL [13]). A service can contain multiple operations on the WSDL level. The terms 'service' and 'service operations', however, are often used synonymously when referring to an operation on the WSDL level. According to Martin [42], there are two classes of Web services. They can be information-providing, e.g. providing flight schedules, world-altering, e.g. placing a booking for a particular flight, or both.

1.1.2 Service-Oriented Architectures

Service-Oriented Architecture (SOA) has evolved as an architectural style in software engineering. SOA is an abstract notion defining an interaction scheme for three loosely coupled entities in the role of either the 'service provider', the 'service requestor' or the 'service registry' (see figure 1.1).

The service provider offers services and descriptions of the provided functionality. These descriptions can be published in a service registry. A service requestor can browse the service registry for service descriptions in order to find service providers offering the demanded service. The service requestor then directly invokes the demanded service at the service provider. All these entities are loosely coupled com-

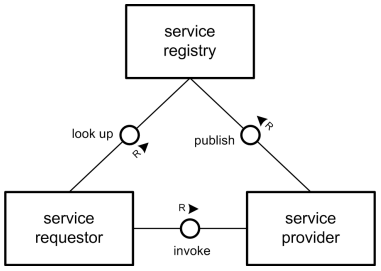

Figure 1.1: Roles in the Service-Oriented Architecture

puter programs which act on the orders of their respective owners.

Web services are the most prominent implementation of a service-oriented architecture. Interfaces are described in the Web Services Description Language (WSDL) [13]. Message exchange between service providers and requestors is built on standards like SOAP [25], HTTP [49] and XML[11]. To support the role of the service registry, the Universal Description, Discovery and Integration (UDDI) [10] standard has been developed.

1.1.3 Web Service Composition

In order to overcome the aforementioned problems, different proposals have been put forward to enhance the conventional Web services stack with a layer that contains process descriptions. Sun, SAP, Intalio and BEA jointly proposed the Web Services Choreography Interface (WSCI [5]). The underlying idea here is to create process

definitions using the Business Process Management Language (BPML), from which a WSCI specification can then be derived.

However, the approach proposed by IBM, Microsoft and, again, BEA, the so-called Business Process Execution Language for Web Services (BPEL4WS) has succesfully outpaced WSCI. BPEL4WS reflects in many ways the proposals of the same vendors, such as WSFL and XLANG. BPEL4WS is now maintained by the OASIS group and filed under the name 'WS-BPEL' [19]. The WS-BPEL approach will be briefly described in the following.

WS-BPEL is an XML based language that allows for modeling the behavior of Web services in a business process interaction. It provides the necessary control structures to express these interactions. These control structures are similar to those of common process modeling notations. They include `Sequence`, `Switch` to support conditions, `Pick` to support events, `While` to support loops and `Flow` to support parallel threads of execution.

WS-BPEL is used to model the behavior of both executable and abstract processes. An abstract process describes the publicly visible interaction protocol between the parties involved by specifying their message exchange. As the interaction is described from the perspective of a single participating service, abstract WS-BPEL specifications do not encode 'choreographies'. The executable process focuses on one specific party and essentially models a private workflow. A workflow engine is necessary to enact WS-BPEL specifications based on that workflow. The executable process contains everything which was left undetermined in the abstract process, such as branching conditions, data assignments, and data transfer rules [4]. The executable part of a WS-BPEL specification is also referred to as the 'orchestration' of the composition.

Throughout this thesis, the terms 'service composition' and 'business process' will repeatedly be used synonymously. The same applies to the terms 'service operation' and 'activity'.

1.1.4 Ontologies

According to Gruber [24], an ontology is 'a formal explicit specification of a shared conceptualization'. Ontologies describe such sets of common terms with taxonomic hierarchies of concepts, as well as their relationships among each other. They describe conceptual dependencies and form common vocabularies that can be shared and agreed upon. The concept of ontologies was developed in Artificial Intelligence as a means to share and reuse knowledge. Ontologies define formal semantics for information which can be processed by a computer. These semantics are real-world semantics in that their contents have a meaning to humans even though they are machine-processable.

The Semantic Web [9] uses ontologies as a key enabling technology to overcome the limitations of syntactic interface descriptions, i.e., WSDL. Syntactic descriptions of Web service interfaces are not sufficient when computer programs, 'agents', are to reason about the capabilities of Web services. In the context of an online book shop,

for instance, a Web service operation can return an ISBN number. A human knows how to interpret ISBN numbers, but to a computer they mean nothing more than strings. Therefore, a layer containing semantic descriptions has to be added on top of the syntactic description of Web services. These semantic descriptions define a vocabulary for the specific domains, e.g. for the book market.

1.2 Goals and Structure

The purpose of this thesis is to investigate the usefulness of semi-automated service composition as a methodology for modeling business processes. Therefore, the mixed initiative functionality that is characteristic for semi-automated composition is to be identified and defined.

The goals of this thesis are as follows:

- The practical applicability of semi-automated service composition is to be demonstrated using a scenario from business reality.
- Semi-automated composition builds upon semantic descriptions of Web services. Therefore, the state of the art in formal semantic service descriptions is to be presented and evaluated.
- While semi-automated service composition is a currently heavily researched topic ([52, 59, 34, 27, 53]), there is no overview of existing approaches available, nor do we have a common understanding of the functionality that is characteristic for semi-automated composition. Therefore, a detailed of related work in the field of semi-automated composition is to be presented.
- As the main contribution of this thesis, three mixed initiative features for semi-automated service composition are to be elaborated. As a first feature, the editor should be able to show all available services that could possibly follow the currently selected activity, i.e. a service, in a business process. As a second feature, the editor should be capable of suggesting sequences of services that connect two activities in a business process, and as a third feature, the modeling tool should check the validity of the semantics of the business process. Here, a business process is considered valid if the activities in the process do not have open information requirements and are not redundant.
- A formal model for service compositions is to be introduced, followed by formal definitions of the proposed mixed initiative functionality.
- To demonstrate the feasibility and the industrial relevance of the presented approach, a prototypical implementation in the context of SAP's Enterprise SOA is to be developed and discussed.
- Specific algorithms for the realization of the three mixed initiative features are to be developed.

The thesis is organized as follows: Chapter 2 provides a general overview of the field of Business Process Management (BPM) as well as a summary of SAP's efforts in this field. It will be discussed what BPM technologies are currently offered or used by SAP. We will also discuss the main concepts of Enterprise SOA and the related efforts to develop a business process platform. Chapter 3 summarizes the state of the art in semantic descriptions and semi-automated service composition. Various possibilities to semantically specify services are presented, discussed and evaluated. Related efforts in the field of semi-automated composition are also discussed. In chapter 4, we present a scenario, which was created on the basis of an SAP product. This business scenario will, then, be used in chapter 5 to introduce the three mixed initiative features. We introduce a rigorous formal model on which these features, as well as possible extensions, are defined. Chapter 6 gives a detailed evaluation of the related work in semi-automated composition that is presented in chapter 3 according to the mixed initiative features and additional criteria. This leads to a set of requirements for a semi-automated modeling environment which is discussed in chapter 7. Architectural considerations for the realization of semi-automated modeling environment are then discussed on the basis of our prototypical realization. Then, the necessary algorithms for the realization of all the three mixed initiative features are developed and discussed. Chapter 8, finally, describes how the presented approach for semi-automated composition could be introduced at SAP. We describe at what points their service-enabling process should be adapted so that a methodology for creating the necessary semantic specifications can be put in place. Chapter 9 concludes the thesis.

2. Business Process Management

The purpose of this chapter is to give a brief overview of Business Process Management (BPM) and what this term comprises. We will describe how SAP has been adopting BPM so far. Therefore, an overview of the methods used within SAP to model business processes will be given. Furthermore, this chapter will introduce Enterprise SOA as SAP's strategy to position the company on the BPM market as well as the key concepts of Enterprise SOA as a basis for the following chapters.

2.1 BPM in General

As a wider trend in IT systems in general, we observe that software systems have become more and more process-aware, in the sense that process logic is separated from application logic. This separation leads to an enhanced flexibility in responding to changes of the business (process) requirements of applications. As the business process is made explicit, it can be changed without redesigning the application. Business Process Management systems facilitate this decoupling of process and application logic. Their purpose is to support business processes in a company either fully or in parts with the aid of software. According to van der Aalst, ter Hofstede and Weske, a Business Process Management system is 'a generic software system that is driven by explicit process designs to enact and manage operational business processes' [68].

Business Process Management (BPM) is currently attracting much attention from both academia and industry. BPM has its roots in workflow management, an area that has emerged in the 1990ies. According to the Workflow Management Coalition (WfMC), a workflow management system defines, manages and executes workflows through the execution of software [30]. The order of execution of the involved activities is driven by a computer representation of the workflow logic. Workflows are, thus, business processes that are being executed with the help of software systems. Workflow management systems provide support for three functional areas: First, they allow for defining the process flow between the activities in the workflow. Second, they manage the execution of workflow processes and ensure a proper sequencing of

the participating activities. Third, they trigger the necessary interactions with human users and applications that are associated with the individual process steps. To summarize, we can say that workflow management systems are primarily concerned with the execution of activities in a predefined, but revisable order called workflow.

BPM in contrast addresses a broader scope than just the enactment of workflows. It comprises methodologies, modeling techniques and tools to define, simulate and validate process flows, as well as mechanisms for their enactment. Mechanisms for monitoring business process during their execution and techniques for the analysis of data that has been gathered during their execution (i.e., process mining) are also in the scope of BPM. Such data can be used for diagnostic purposes and can serve as a starting point for process optimization. Sub-disciplines of BPM such as Business Process Analysis (BPA) and Business Activity Monitoring (BAM) have evolved to focus on the diagnosis step of the BPM lifecycle. The BPM lifecycle, also showing the overlapping of workflow management and BPM, is depicted in figure 2.1.

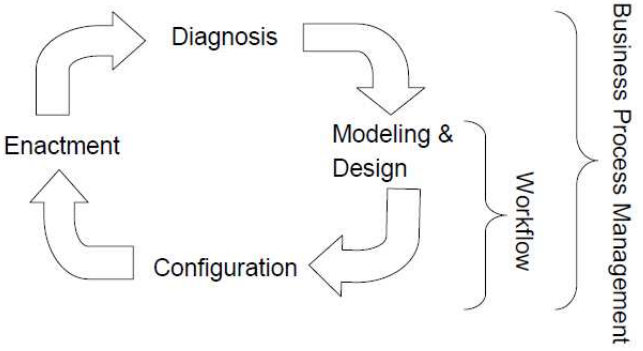

Figure 2.1: The BPM lifecycle (from [68])

2.2 BPM at SAP

The purpose of this section is to describe SAP's approach towards BPM. We will, therefore, discuss how processes are modeled today with the aid of SAP software. This will lead to the motivation and introduction of Enterprise SOA, a new paradigm for the reuse of functionality provided by SAP.

2.2.1 Modeling tools

SAP provides a multitude of different tools that can be used for service composition. In the following, these tools will be briefly described. We will see that while all the tools have different scopes, most can be used to integrate enterprise services into processes to some extend.

2.2.1.1 Guided Procedures

Guided Procedures is a tool for frontend process orchestration, in the sense that it aims at composing process flows out of user interfaces. Hence, it is scoped for the creation of conversational and user-centric services, which are types of processes

that appear repeatedly in user interfaces. Basically, a Guided Procedure walks a user through a series of steps that may occur across multiple applications. The user's progress throughout the progress is visualized. The steps can consist either of portal-style user interfaces or Portable Document Format (PDF) forms which are filled out directly in the browser. In both cases, the fields in the user interface are mapped to database fields in an arbitrary SAP system. Besides the user interfaces ('blocks'), there is the notion of 'callable objects'. This concept allows the execution of an enterprise service as a backend function for the respective process step. Guided Procedures are modeled with tools which are part of the Composite Application Framework (CAF).

2.2.1.2 ccBPM

Cross-Component Business Process Management (ccBPM) is a tool that belongs to SAP NetWeaver Exchange Infrastructure (XI). It is used for automating backend processes, which are types of processes that require minimal or no user intervention and, typically, take a long time to complete. When enacted, the processes are regarded as 'transactions'. The transactions run automatically and are driven by events which are sent from other transactions via XI. ccBPM can route, map, and process messages sent at high volumes, while it keeps track of the different transactions. The transactions can span multiple applications. The actual activities in the process models can be linked to XI interfaces. It is therefore possible to build service compositions consisting of enterprise service operations using ccBPM. It is possible to export such service compositions in the WS-BPEL [19] format.

2.2.1.3 Visual Composer

Visual Composer is a Web browser based modeling environment for user interfaces. It is based on HTML, JavaScript and GML script, and is shipped together with SAP NetWeaver. Visual Composer can currently be used for modeling user interfaces. In the next release of NetWeaver, however, it will be possible to model Web service compositions representing business processes. The user interface modeling capabilities will still remain in the product to allow for the easy creation of user interfaces for supplying the Web service parameters. However, the process modeling capabilities of Visual Composer are limited, as it will only support user-interface driven processes. What's more, the graphical modeling language is non-standard and lacks the flexibility of languages which are specifically designed for modeling business processes, e.g. the Business Process Modeling Notation [69] (BPMN). It is also planned to enhance Visual Composer with the possibilities to create Guided Procedures.

2.2.1.4 ARIS

The ARIS Business Designer from IDS Scheer is used to describe high-level structures of business processes. It is used for the model-driven process orchestration of Enterprise SOA processes. At the highest level, the process is divided into process components. The modeler has to model three different aspects of the process that is being described: First, the landscape of process components is described in

a so-called 'integration scenario'. Second, the orchestration of the individual process components is described by modeling how enterprise services are modeled on top of business objects[1]. This step is called 'process component modeling'. Third, the choreography of the process components, i.e. their interactions is described in 'component interaction modeling'. The process models created in ARIS are directly executable, but are also used to explain a business process to other information workers.

2.2.1.5 Composite Application Framework

For reasons of completeness, the Composite Application Framework (CAF) itself could be considered a process modeling environment. In contrast to frontend and backend process orchestration, the use of CAF without additional toolsets would result in a form of 'code-level process orchestration'. The developer has the possibility to build processes out of enterprise services by writing ABAP or Java code which encodes the process flow among the consumed enterprise service operations. Such a process can, then, again itself be exposed as a composite service. It is an interesting fact that most of the compositions of enterprise services are built directly in CAF, showing the need for a comprehensive modeling tool for that case. Encoding process flow in low-level code is, obviously, a rather inflexible approach to create service compositions.

2.2.1.6 Conclusion

We have described different tools that can be used to create compositions of enterprise services. All tools are maintained separately from each other and that there is no common modeling methodology. The Business Process Modeling Notation (BPMN) [69] as the emerging de-facto standard for high-level business process modeling is incorporated in none of the tools presented. In contrast, BPMN has been adopted by some of SAP's competitors in the BPM market. It is used, for example, in tools such as WebSphere Business Modeler from IBM and Intalio|Designer from Intalio.

We have seen that none of the presented modeling tools provides the user with guidance in assembling composite processes: In Guided Procedures, there is no mechanism that helps the user to deal with the sheer complexity of available callable objects. Guided Procedures is a completely static mechanism for manual service composition. The same holds true for ARIS Business Developer. ccBPM is a manual modeling tool which is intended for developers. It does not provide any mixed initiative functionality to aid the user in the creation of service compositions. Visual Composer cannot be used to build executable processes.

2.2.2 Enterprise SOA

SAP is a supplier of business applications which provide IT support for core business processes within enterprises. Specifically, SAP today offers a suite of packaged applications that mainly support traditional business processes which are usually

[1] The concept of 'business objects' will be explained in section 2.2.3

not subject to change. These packages form the mySAP Business Suite. The core solutions within that suite comprise Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), Product Lifecycle Management (PLM), Supply Chain Management (SCM) and Supplier Relationship Management (SRM).

These packaged applications can, of course, only encode best practices. As a result, the customers must adhere to the procedures provided by the SAP applications. If the SAP system has to be adapted to support specific processes in a company, costly customization efforts are required. Therefore, differentiation through their core business processes is difficult for the customers. Now, SAP has identified the need to give their customers the flexibility to adapt the shipped business processes to their specific needs and, also, to create new business processes which are of particular strategic importance. In doing so, the customers will be able to run business processes that differentiate their company amongst others with the help of an SAP system. As of today, business processes that are of strategic importance and are subject to change are, mostly, still implemented in custom software development efforts by the customer.

In order to allow for business process flexibility, the functionality provided by the enterprise software systems must be broken down into fine-granular building blocks. At the same time, the functionality must not be offered on a too fine-granular level of abstraction so that the complexity resulting out of the number of available building blocks can still be handled. SAP plans to deliver their functionality as single operations that reflect business process steps, instead of providing them through monolithic software packages. These coarse-grained operation steps, which are called enterprise services, are grouped around business processes. Enterprise services provide a meaningful business-level operation. They are intended to be understandable by users without technical background, such as business analysts. Due to the fact that the enterprise services will be exposed and accessed as Web services, this strategy was named Enterprise SOA, strongly resembling the notion of Service-Oriented Architectures (SOA).

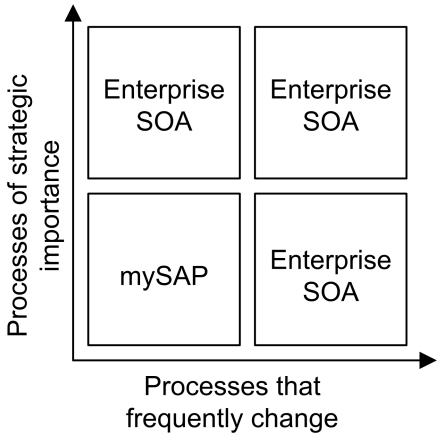

Figure 2.2: Types of business processes supported by SAP systems

Figure 2.2 depicts the positioning of Enterprise SOA according to the types of business processes that are to be supported: On the one hand, the mySAP suite supports

business processes that are unlikely to be of strategic importance for a company, as they deliver solutions for common business scenarios. These processes are also unlikely to be changed very often, and a company does, thus, not need the flexibility to change them. On the other hand, Enterprise SOA allows a company to create business processes that differentiate a company from its competitors, i.e. strategic processes. Enterprise SOA is also an enabler for processes that are frequently subject to change and are thus required to be flexibly supported by IT. Besides delivering the enterprise services, SAP will also provide the platform for enacting the business processes. The current SAP NetWeaver platform will, consequently, evolve into a complete BPMS. This BPM suite, marketed as the SAP Business Process Platform (BPP), will allow the specification, enactment and monitoring of enterprise service compositions.

2.2.3 Conceptual foundations of Enterprise SOA

SAP's enterprise services are technically based on Web services. Specifically, WSDL [13] is used for describing their interfaces and XML Schema (XSD) [6] for the relevant data types. From a logical perspective, the so-called 'business objects' are seen as the service providers of the enterprise services. Business objects are the primary structuring elements within Enterprise SOA. They are units of application logic which are grouped around entities that are fundamental for business applications. Examples of business objects are 'sales order', 'invoice', 'customer' or 'business partner'. While the notion of an object in the object-oriented world refers to an instance of a class, business objects in Enterprise SOA reside on the type-level. Thus, the business object *invoice* represents the set of all invoice instances.

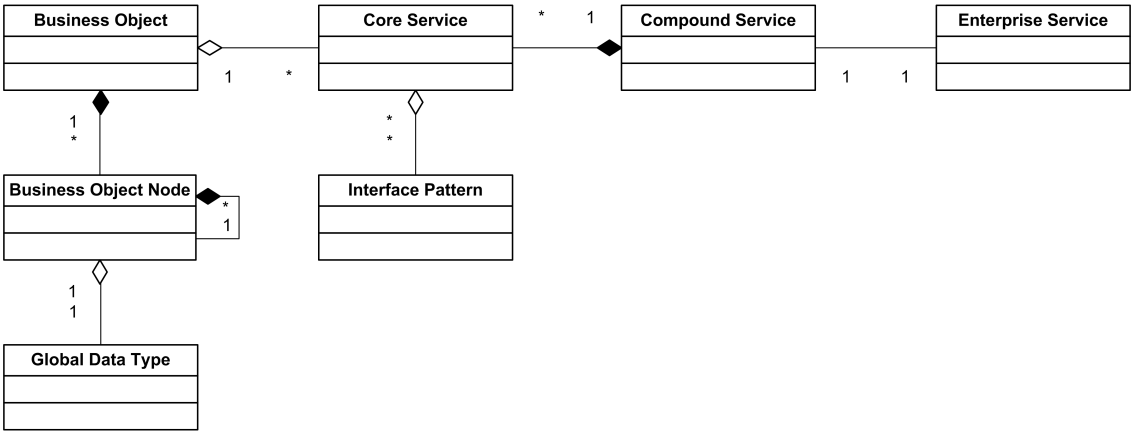

Figure 2.3: Enterprise SOA Metamodel

Figure 2.3 depicts the logical metamodel of Enterprise SOA. The business objects, defined in the Enterprise Service Repository (ESR), are trees of business object nodes. A business object node is structurally defined by a Global Data Type (GDT). In other words, a business object is a structured set of GDTs. These are SAP-wide defined and consolidated data types. They are compliant with current business data integration standards. Specifically, the GDTs have been created according to the Core Component Type Specification (CCTS) [15] modeling methodology, defined

by the UN/CEFACT council. The advantage of the CCTS methodology is that it distinguishes between data types on a generic level (core components) and data types for specific vertical industry. This distinction allows to have a generic data type 'sales order', which can, for example, be extended specifically for the use in the oil and gas industry. CCTS does not only distinguish different vertical industries, but also supports business process, product classification, system capabilities and geopolitical contexts. CCTS components in a specific context are referred to as Business Information Entities (BIE). According to the Enterprise SOA metamodel, it is possible to create complex business objects through composition. For example, the business object *sales order* is composed out of *sales order item* business objects. Associations between business objects are also possible. So, the business object node *buyer party* of a *sales order* could be associated with the business object *business partner*.

Business objects provide a set of 'core services' through which they can be accessed by other business objects or services. Through standardized interface patterns, the operations of the core services and their parameters can be derived. The 'access' pattern, for instance, defines the core operations *create*, *retrieve*, *update* and *delete* (amongst other operations). Other interface patterns allow for querying a business object for business object node data based on given selection criteria, executing an action, handling transactional contexts and retrieving value sets. It is noteworthy that all these different operations, provided through the interface patterns, result in separate service operations on the WSDL level. On top of the core services, there are 'compound services' that provide a functionality on a higher level than it is delivered through the interface patterns of the core services. Such services characteristically act on multiple business objects which are semantically related. They are required to invoke core services of the business objects that they use in order to provide their functionality. Finally, the 'enterprise services' are a subset of the compound services. According to [21], an enterprise service is a compound service which is used in the execution of a business process step, having a significant meaning and impact for the business of an enterprise.

3. State of the Art in Semantic Descriptions and Semi-Automated Composition

The purpose of this chapter is to present the state of the art in semi-automated service composition. First, semantic descriptions for Web services will be discussed as a preliminary technology. It will be discussed which kinds of semantic descriptions and ontologies are necessary for enterprise services so that they can be used for a semi-automated service composition. Second, we will investigate the most prevalent approaches for capturing service semantics and provide an evaluation.

3.1 Semantic Descriptions

As we have seen in section 2.2.3, the interfaces of enterprise services are published in the WSDL [13] format. This section deals with semantic descriptions that can be applied to Web services in general. We will present the most important approaches to modeling Web service semantics at the time of this writing, OWL-S, WSMO, SWSF, WSDL-S and SAWSDL. These approaches will, then, be compared and evaluated according to their usefulness in the context of this work.

3.1.1 OWL-S

The DARPA Agent Markup Language (DAML) Program, which was set up in August 2000, has the purpose of developing a language and tools to facilitate the concept of the Semantic Web. After the development of DAML+OIL in 2000 and DAML-S in 2001, the efforts of the program are, currently, centered on the development of OWL-S. OWL-S is a framework for semantically describing Web services. It is based on the Web Ontology Language (OWL), which was also developed by the DAML group. OWL was developed as a vocabulary extension of RDF (the Resource Description Framework) and was derived from the DAML+OIL Web Ontology Language. There are three increasingly expressive sublanguages: OWL Lite, OWL DL and OWL Full. OWL Lite has a limited expressiveness and was designed for beginners to create simple ontologies. It was also designed for easy implementation, for only a limited

subset of the Web Ontology Language is available. OWL Full and OWL DL support the same set of OWL language constructs. OWL DL is more restrictive concerning the usage of these language constructs, i.e. it requires pairwise separation of classes, properties, individuals and data values, as well as the usage of RDF elements in OWL. It offers computational completeness and decidability. In OWL Full, all language constructs can be used without restrictions and free mixing with RDF Schema, but there are no computational garantuees [17]. Thus, the expressivenesses L of the three sublanguages are related as follows:

$$L(\texttt{OWL Lite}) \prec L(\texttt{OWL DL}) \prec L(\texttt{OWL Full})$$

An 'OWL-S ontology' consists of three main parts: The service *profile*, the *process model* and the service *grounding*. It should be noted that the term 'ontology' is normally used to describe the concepts and the taxonomy to which a semantic services description relates. In OWL-S terminology, however, the semantic service description itself is called an ontology. The OWL-S ontology is depicted in figure 3.1.

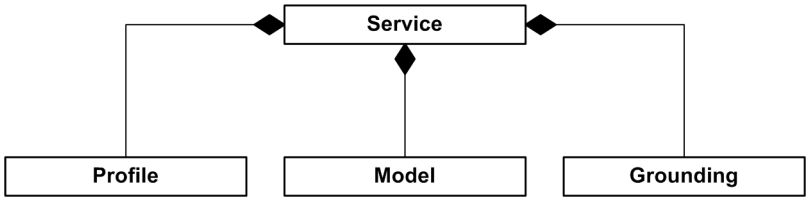

Figure 3.1: Top level of the OWL-S service ontology

The profile describes 'what a service does' [42]. The data transformation which is accomplished by the service is characterized by *inputs* and *outputs*; the state of the world before and after the execution of the service is described by its *preconditions* and *effects* (if any). Abbreviated, this information is called the IOPEs of a service. In general, OWL classes are used as parameter types for the inputs and outputs. They are referred to by an URI which, usually, identifies an element in an ontology. The preconditions and effects are described as logical formulae, which can be stated in logic languages like KIF or DRS. This allows software agents to reason about whether a service is appropriate for a specific task or not, given that the service profile can be found in a registry the agent can access. Furthermore, the service profile provides information about the organization providing the service, such as a contact person and address. Also, a host of properties describing the features of a service can be offered. Here, the category of the service is specified corresponding to an arbitrary classification system as well as to a quality rating of the service and a list of service parameters that can provide any information (e.g., the maximum response time of the service).

The OWL-S process model specifies the ways in which a client can interact with the service. OWL-S distinguishes *atomic* and *composite* processes: Atomic processes expect one message and return one after they have been performed. Note that these messages can have an arbitrary number of parameters. In contrast, composite

processes are stateful in such a way that the client has to interact with them several times if the whole process is to be performed. For composite processes, it is necessary to describe their decomposition through control structures like sequence, if-then-else, split, etc., which are provided within OWL-S. The data flow between subsequent steps of a composite service is also specified in the model.

If we consider the example of an online book store, there could be three atomic processes, e.g. searching for an author, adding a book to the shopping cart or paying for a book. The OWL-S process model would then allow us to state that the whole process of shopping for books online is a sequence of the three atomic processes.

The process model describes local parameters and allows expressing conditions for outputs and effects which are grouped as 'results'. This is necessary as services do not always have exactly one result. In a buying process, for example, the effect would be that the buyer's credit card is charged and the ownership of the purchased items is transferred to the buyer. This result must not occur in case the precondition that the buyer must hold a valid credit card is not met. In this case, the process should result in the output of an error message. The process model should be consistent with the profile, because the client uses the process model to learn how to interact with the service, after the service has been selected according to the transformations described in the profile. If this is not the case, communication will be interrupted at some point. Yet, OWL-S does not specify any constraints between profile and process, so that any inconsistency between the two does not affect the validity of the OWL expression [42].

The third element of an OWL-S ontology is the service grounding. Groundings provide details on how the service can be accessed, such as protocol and message formats. It can be seen as the linking pin between an abstract specification and a concrete realization. In theory, any kind of grounding is conceivable, but, due to industry adoption, OWL-S ontologies are usually grounded to WSDL-specifications. *Atomic processes* in OWL-S and *operations* in WSDL overlap, as do *inputs* and *outputs* in OWL-S and the *message*-part of WSDL. One has to provide a mapping between operations in WSDL and atomic processes in OWL-S. It is possible to provide an XSLT stylesheet for the grounding that specifies this mapping. Also, the input and output parameters on the syntactical layer (the WSDL messages with their XML Schema data types) must be mapped to inputs and outputs in the sense of OWL-S, referencing the appropriate classes in an ontology. This can be achieved by extending the WSDL elements *types*, *message*, *operations* and *binding*, which can be done by the use of so-called WSDL extensibility elements without having to revise the base WSDL specification.

3.1.2 WSMO

The Web Services Modeling Ontology (WSMO) [54], which is being developed by the WSMO working group, constitutes another framework to conceptually model Web service semantics. It is build upon and extends the Web Service Modeling Framework (WSMF). WSMO distinguishes four top level elements: *Ontologies* are

used to share terminology, descriptions of *services* which are requested or provided, *goals* that specify problems to be solved by the Web services and *mediators* that somehow link the different WSMO elements. These elements are now to be discussed in greater detail.

Ontologies have already been discussed as a main technology for the Semantic Web. But how are ontologies represented in WSMO? First of all, they can be attributed - like almost any other element in WSMO - with so-called nonfunctional properties. Most of these properties are taken from the Dublin Core Metadata Element Set [12]. Ontologies in WSMO can be attributed with information about the publisher, owner, their subject (usually expressed in keywords), their coverage, i.e. their temporal or regional scope, and much more. A promising concept is the idea that ontologies can directly import other ontologies, supporting modularization is the naïve approach for handling complexity. If such an import raises any conflicts between the elements of the different ontologies, WSMF ontology mediators can be used to resolve them by aligning the imported ontology in a suitable fashion. The actual entities constituting an ontology are called 'concepts' in WSMO. Concepts can have attributes with names and types. They are ordered according to a hierarchy of 'superconcepts' and 'subconcepts', which strongly resembles the notion of classes in OWL. While there are several language constructs in OWL by which relations between classes can be described (disjointWith, inverseOf), all interdependencies in WSMO are represented with either the 'relation' or the 'function' construct. These constructs directly support axiom logical expressions, by which elements and their constituent parts can be modeled in a formal and unambiguous manner. Like the ontology descriptions, all logical expression must be stated in the Web Service Modeling Language (WSML) [16] which is developed and maintained separately from WSMO by the WSML working group. Similar to OWL-S, WSML provides multiple language flavors to maintain different levels of expressiveness. Besides the WSML flavors stated in the following equation, there is also `WSML-DL` which builds on top of `WSML-Core` and compares to `OWL-DL`.

$$L(\texttt{WSML-Core}) \prec L(\texttt{WSML-Flight}) \prec L(\texttt{WSML-Rule}) \prec L(\texttt{WSML-Full})$$

The actual service definition is manifested in the `webService`-part of the WSMO specification. This section describes the capability of a service and its interface. In WSMO, a Web service defines one and only one capability. WSMO describes capabilities through pre- and postconditions, assumptions and effects, each of which are expressed with logical formulae. Pre- and postconditions model the information space of the Web service before and after its execution, whereas assumptions and effects describe the world state. When all preconditions and assumptions of a Web service are met, its execution implies that the respective postconditions and effects become valid. In contrast to OWL-S, the informational state of a Web service is explicitly modeled in WSMO by assumptions and postconditions. OWL-S limits itself to describing the transformation of inputs into outputs statically in terms of concepts contained in the used ontology. Besides the capability of a Web service, the WSMO service definition also describes its interface: The choreography and the

orchestration are specified as state machines with guarded transitions, opposing the workflow-based approach observed in OWL-S. WSMO does not support composite services.

WSMO introduces the concept of *goals* to represent the objectives of users when consulting a Web service. Goals are, usually, subsets of Web service capabilities which are of particular interest for the client (i.e., postconditions or effects). According to Hutter, this reflects the so-called goal driven approach in AI planning [31]. WSMO argues for the necessity of goals by stating that this approach was, naturally, more focused on the client, as goals describe what a user can expect from a service rather than what the service does.

The linking pin between the different constituents of a WSMO description are *mediators*. They resolve conflicts between ontologies, translate between outputs and inputs of connected Web services, link capabilities to goals or declare goals as substitutable by each other. Therefore, WSMO defines four different types of mediators, namely `ggMediators`, `ooMediators`, `wgMediators` and `wwMediators`.

The messages used to interact with a Web service contain XML Schema-typed parameters. WSMO, in contrast, describes services on a conceptual level using ontologies. Naturally, a mapping between the ontology concepts and XML schema is necessary. WSMO proposes three possible approaches to realize this mapping. Their preferred approach is to create mappings on the conceptual level [36]. Here, WSMO ontologies are automatically created from XML Schema definition in an ad-hoc manner. These are, then, linked to a WSMO ontology which conceptually describes the service using existing WSMO mediation tools. This results in a two-level rule set that can be applied at runtime. A second way is to use XSLT to create a direct mapping between the semantic- and the syntax level description of a service. As a third approach, a direct mapping between the source XML data and the target WSMO ontology can be established. Therefore, a mapping language which was specifically developed for this purpose (e.g., WSDL-S [3] or SAWSDL [20]) is to be used.

3.1.3 SWSF

The National Institute of Standards and Technology (NIST), National Research Council of Canada, SRI International, Stanford University, Toshiba Corporation, and the University of Southampton propose Semantic Web Services Framework (SWSF) [7] as a W3C member submission.

SWSF comprises two components: The Semantic Web Services Ontology (SWSO) and the Semantic Web Services Language (SWSL). SWSO presents a conceptual framework for describing Web service capabilities. It is based on an axiomatization in first-order logic to describe the concepts in the ontology. SWSL is the first-order logic language that underlies the SWSF approach. The authors describe it as a general-purpose logical language that comes with certain additional features to make the language usable in the context of Web services. SWSL comes in two sublanguages of different expressiveness, namely SWSL-FOL and SWSL-Rules. SWSL-FOL is a first-order logic, while SWSL-Rule is a fully-fledged logic programming language.

According to [7], nearly all elements of the syntax are common to both SWSL-FOL and SWSL-Rule. SWSL-Rule is more expressive than SWSL-FOL.

$$L(\texttt{SWSL-FOL}) \prec L(\texttt{SWSL-Rules})$$

The focus of SWSF is put more strongly on providing an ontology with a well-defined theoretical semantic than on providing an executable specification. The authors state that according to their experience, most AI reasoning is done by special-purpose reasoners and do not further discuss eventual decidability problems when reasoners are to exploit the specifications. SWSO also allows for modeling choreographies and abstract process models. Therefore, a variety of control constraints are offered, covering most of the basic workflow patterns. It is also possible to attribute services with a predefined set of nonfunctional properties on the service level (i.e., it is not possible to state nonfunctional properties for every operation that a service provides).

3.1.4 WSDL-S

IBM and the University of Georgia jointly propose Web Service Semantics - WSDL-S [3]. The authors argue that prevalent proposals to enhance Web services with semantics, such as OWL-S and WSMO, are not aligned with existing standards. In contrast, WSDL-S proposes to enhance the commonly adopted WSDL standard, whereas OWL-S and WSMO largely replace WSDL, as these frameworks include their own service descriptions. These descriptions can be grounded to WSDL interfaces, but are designed independently of WSDL. The proposal of WSDL-S is, thus, to include semantic annotations directly in WSDL, while referencing external ontologies. As ontologies are decoupled from the WSDL-S approach, it is possible to use any arbitrary ontology description language. The proposed semantic annotations to be added to WSDL compare strongly to the service profile in OWL-S: A service is characterized by inputs, outputs, preconditions and effects, where these terms carry the same semantics as the IOPEs described in section 3.1.1. In order to annotate services with this information, WSDL-S makes use of the extensibility elements of WSDL. WSDL-S only enhances the abstract part of the WSDL specifications, namely *types* and *operations*. As far as the types section is concerned, two extension elements are provided: The `modelReference`-attribute maps XML schema types to concepts in a semantic model in a one-to-one manner. The `schemaMapping`-attribute does the same, but describes one-to-many or many-to-one mappings. This compares to the inputs and outputs in OWL-S service profiles. WSDL's operations section is extended by the `modelReference`-attribute as well as bytwo elements called *preconditions* and *effects*. The latter ones either refer to the part of an external ontology specifying the actual precondition or effect, or directly include expressions which describe them. In the latter case, the formatting of the expressions depends on the utilized language to model semantics. Additionally, WSDL-S proposes an extensibility element `category` by the aid of which Web services can be categorized according to a taxonomy.

3.1.5 SAWSDL

The most recent approach to specify Web service semantics is Semantic Annotations for Web Services Description Language (SAWSDL) [20]. The W3C founded the SAWSDL working group in March 2006 with the intention to develop 'a standard solution for Web automation' [65]. The SAWSDL working group is part of the Web Services and Semantics (WS2) Project, financed by the European Commission's IST Programme (EC IST FP6).

The SAWSDL approach is conceptually similar to the WSDL-S approach, which was presented above. SAWSDL solely builds on the extensibility elements in the WSDL 2.0 specification. The idea is to directly annotate the abstract part of WSDL interfaces with semantics. In detail, the proposal suggests the introduction of three extensibility elements:

- A `modelReference` attribute to link a WSDL element (i.e., complex types, elements and operations) to a concept in an ontology.
- A `schemaMapping` attribute to map XML Schema data types to concepts in an ontology, capturing possible structural differences between a complex data type in WSDL and a concept hierarchy in an ontology.
- A `category` element that allows to specify a service category according to an arbitrary categorization scheme.

These are the same extensions that are proposed by WSDL-S, and they even carry the same names. In contrast to WSDL-S, SAWSDL does not propose elements for the specification of preconditions and effects.

3.1.6 Evaluation

Throughout the preceding paragraphs we have presented the most prevalent approaches towards semantic service descriptions by the time of this writing. They are all very similar, as basic concepts seem to recur in all of the frameworks. Yet, all have different scopes: While OWL-S, WSMO and SWSF are frameworks that provide languages to model ontologies instead of limiting themselves to describing services, WSDL-S and SAWSDL are agnostic of possibly utilizable languages. In contrast, they put a stronger emphasis on the practical ease with which to deploy descriptions, as their approach solely bases on extensibility elements of WSDL. However, these approches can be useful when WSDL specifications for services are already available before the semantic descriptions are modeled. It is still possible to link the annotated WSDL files to OWL-S or WSMO ontologies and to draw upon existing reasoners for these languages.

Still, WSDL-S and SAWSDL cover a rather small scope, which makes them difficult to compare to OWL-S, WSMO or SWSF: Many distinctive features depend on the ontology language and the larger context of the framework. Table 3.1 introduces

	OWL-S	**WSMO**	**SWSF**	**WSDL-S and SAWSDL**
Number of interfaces per service	Exactly one	Multiple	Exactly one	Exactly one
Degrees of expressiveness of ontology language	3 layers	5 layers	2 layers	N/A
Languages for logical expressions	SWRL, DRS, KIF	WSML	SWSL	N/A
Functional service description	Capability-driven	Goal-driven	Capability-driven	Capability-driven
Presence of nonfunctional properties	Present in profile	Present in all elements	Present in service descriptor	Not present
Format of nonfunctional properties	Non-standard	Partially standardized	Non-standard	N/A
Modeling of composite services	Workflow-like	Not addressed	Workflow-like	N/A
Maturity level	Medium	Low	Low	Low
Tool support	Largely available	Sporadically available	Sporadically available	Sporadically available
Support for modularization	Simple ontology import	Mediated ontology import	Simple ontology import	N/A
Availability of predefined ontologies	Largely available	Hardly available	Hardly available	N/A
Available reasoners	Many / mature	Many / immature	Few / immature	N/A

Table 3.1: Evaluation of OWL-S, WSMO, SWSF, WSDL-S and SAWSDL

distinction criteria for Semantic Web service frameworks by which approaches like OWL-S, WSMO and SWSF can be compared.

In OWL-S, a service has exactly one service model, which means that there is exactly one way to interact with it. The same holds true for SWSF, WSDL-S and SAWSDL, as extensions of WSDL, also offer only one interface, i.e. the corresponding WSDL specification. In WSMO, the choreography is part of the Web service interface, and a service can have multiple interfaces. This can be seen as a clear advantage of WSMO, as it allows for greater flexibility in the choreography of Web services.

OWL-S is based on the Web Ontology Language (OWL) which provides three different flavors with different degrees of expressiveness. Each of these is scoped to provide the modeler with a reasonable tradeoff between ease of use and expressiveness. WSML features five different flavors, which is a larger number than the three languages presented by OWL-S. SWSL comes in two different flavors. Still, the advantage in terms of language flavors is questionable, as too much choice results in higher complexity. The tradeoff with languages of a high expressiveness is that decidability cannot always be guaranteed. The decidability depends largely on the complexity of the domain model, so that neither of the languages has an advantage here.

While OWL-S only enforces the use of OWL to specify the domain models, it leaves the choice of the language used to express logical conditions to the user. OWL-S, currently, supports the languages SWRL, DRS and KIF. WSMO restricts the user to use WSML for stating logical expressions. SWSF, similarly, restricts the user to use SWSL for that purpose.

In OWL-S, WSMO, SWSF and WSDL-S, the capabilities of a service are described by the informational transformations and the change in the world state realized by this service when executed. Only SAWSDL does not allow to model changes in the world state that are incurred by services. WSMO argues that the user is particularly interested in the state of the information space and the world after the execution of a service. Thus, WSMO describes these as goals, which are a subset of the service capabilities. In doing so, WSMO directly supports what is known as the 'goal-driven' approach in AI planning [31]. This can be helpful for matchmakers to determine the suitability of a service for a specific task. In contrast, a WSMO goal is a subset of a WSMO capability, which makes it redundant information. Therefore, the concept of *goals* in WSMO seems to be questionable.

Nonfunctional properties of services can be expressed in OWL-S, WSMO and SWSF. In WSMO, all constituents of an ontology can be attributed with nonfunctional properties. OWL-S only allows them in its service profile, and in SWSF they can only be applied to the 'service descriptor' (which is similar to the OWL-S service profile). The WSMO approach seems to allow for highest degree differentiation. Furthermore, WSMO proposes the use of accepted metadata standards (Dublin Core [12], FOAF [18]) to present a service's nonfunctional properties. OWL-S does not explicitly propose this, while the use of standard metadata formats is not ruled out. In SWSF, the format of the nonfunctional properties must comply with the underly-

ing first-order logic axiomatization of SWSL. WSDL-S and SAWSDL do not, yet, support nonfunctional properties.

Another difference between the frameworks arises in the way composite services are modeled. OWL-S and SWSF enable the modeler to use well-known constructs from the area of workflow management, whereas WSMO is, currently, not able to represent compositions. A service's choreography can be described using state machines with guarded transitions in WSMO. While WSDL-S and SAWSDL do not support composite services at all, because they operate on the WSDL level of a service description, OWL-S seems to present the most convenient approach.

OWL-S is a more mature technology than WSMO, WSDL-S and SASWDL, which is probably due to the fact that OWL-S has been developed for a longer time than the others. This is also a result of its user acceptance and, thus, its practical appliances which are also a source to feedback for the developers of OWL-S. Since OWL-S has been available for a longer time, there is also more tooling around which complements the framework, e.g. modeling tools for ontologies. Tool support is also increasing for WSMO, while there are, currently, no tools available for SWSF.

WSMO addresses heterogeneity issues in service description directly by introducing mediators to map between the different elements of a WSMO specification. OWL-S treats this more like an architectural problem. As it is a likely scenario that ontologies will be developed by different parties (especially for complex domains), it is an advantage that WSMO attempts to address the problem how different ontologies that possibly overlap could be integrated.

Nowadays, most of the publicly available domain models are specified in OWL, as a result of its development over a long time. Reusable ontologies for WSMO can also be found on the net. Reusable domain models in SWSF are hardly available at present.

The most important criterion for the evaluation of ontology languages is whether reasoning tools are available to make practical use of the ontologies. We will, therefore, give a brief overview of the available reasoning support for the various languages. As WSDL-S and SAWSDL are agnostic of the ontology that is used along with the service descriptions, we exclude them from the discussion.

3.1.6.1 Reasoning support for OWL-S

A variety of automated planners and inference engines are available for OWL-S, some of which have been under development for more than two years and can, thus, be characterized as mature software.

- Racer [26] is a commercial OWL reasoner and inference server which has been developed by Ralf Möller. Racer provides full support for OWL-Lite and almost full support for OWL-DL. The restrictions are that user-defined data types and nominals, i.e. individuals in class expressions, are only approximated. Racer does not directly support service composition.

- OWL-S Matcher [32] has been developed by Michael C. Jäger and Stefan Tang at the Technical University of Berlin. The matchmaker compares two descriptions (one in form the service requester and another by the service provider) and identifies different relations between the two descriptions (e.g., 'match' or 'no match'). Service composition is also not directly supported.

- Semantic Tools for Web Services [2] is a set of Eclipse[1] plug-ins for the semantic matching and the composition of Web services. It is being developed by IBM Research, as part of their Emerging Technologies Toolkit (ETTK).

- DAML-S Matchmaker [61], which has been developed by Katia Sycara at the Carnegie Mellon University, provides matching of DAML-S profiles. The composition of services is not directly possible with this application.

- SHOP2 [60], which stands for Simple Hierarchical Ordered Planner, has been developed by Evren Sirin at the University of Maryland. It uses a planning algorithm based Hierarchical Task Networks to support the composition of Web services.

3.1.6.2 Reasoning support for WSMO

Several reasoners for WSMO have been developed during 2005. In the year before, no reasoners for WSMO were available.

- WSML Reasoner represents a reasoning environment that is based on the WSML2 Reasoner Framework [28]. The framework allows for translating ontology descriptions from WSML Rule to predicates and rules. Additionally, it provides a facade for easy integration of different reasoners, such as MINS [40] and KAON2 [47].

- WSMX [14], which stands for Web Service Execution Environment, is currently released as version 0.2. It is the reference implementation of WSMO and is supposed to support both discovery and composition of Web services. WSMX is being used and further developed by several EU projects, such as DIP, SEKT and Knowledge Web.

- IRS-III [27], which is short for Internet Reasoning Service, is being developed by the group of John Domingue at The Open University. IRS-III is a platform where semantically described services can be registered and executed. These services can, then, be discovered and composed in the platform.

- Adaptive Services Grid (ASG) [37] is an Integrated Project supported by the Sixth Framework Programme of the European Commission. It comes with a reasoning component that is based on the WSML2 Reasoner Framework. It transforms WSML ontologies into the FLORA-2 [70] language. Additionally, a planning component is availavle that creates service compositions using a heuristic search algorithm and returns the result in WSML.

[1] See `http://www.eclipse.org/`

3.1.6.3 Reasoning support for SWSF

We see SWSO as a theoretical ontology which has no practical impact. The focus lies more on providing a well-formed semantics for modeling ontologies than on having executable specifications. However, one reasoner is available for SWSF at the time of this writing: Vampire [62], which was developed by Xing Tan of the University of Toronto, supports basic queries, such as checking ontology consistency. Web services discovery and composition are not realized.

3.2 Semi-Automated Composition

The purpose of this section is to give an overview of current research efforts regarding semi-automated service composition. Four approaches will be presented with respect to their main characteristics. The evaluation of these approaches is, however, postponed to chapter 6. This separation of presentation and evaluation of the work related to this thesis is necessary, because we, first, aim to introduce the three mixed initiative features for semi-automated composition, before we evaluate related approaches in this field. In doing so, the features can be considered in the evaluation. Furthermore, the thorough evaluation of the related work in chapter 6 will add several aspects to the requirements for the realization of the work presented in this thesis.

3.2.1 Web Service Composer

Sirin, Parsia and Hendler [59] present a prototypical implementation of a composer for Web services. Their tool allows creating executable compositions of Web services that are semantically specified with OWL-S [42].

The created service compositions can, in turn, be stored as OWL-S 'process models'. Process models are a part of OWL-S ontologies which are, normally, used to encode the orchestration for a described service. Well-known control constructs from the area of Workflow Management can be used within OWL-S process models. It is, therefore, a suitable format for representing composed services.

The focus of their work lies on filtering the list of available services at each composition step and thus helping the user to select the appropriate services.

In order to create a composed service, the user follows a backward chaining approach. He or she begins with selecting a Web service that has an output producing the desired end result of the composition from a list of all available services. Next, the user interface presents additional lists connected to each OWL input type of the service producing the end result. In contrast to the first composition step, these lists do not contain all available services: They contain only those services that generate an output compliant to the particular input type they are connected to. An output of a service A is compliant to an input of a service B if their types are exactly the same or if the output of A subsumes the input of B, i.e. the input of B is a specialization of the output of A. If a service is selected from the list of compliant services, this service's inputs must again be produced by selecting services producing

compliant outputs. This is repeated until the user decides, at one point, to provide the inputs that are not connected to a compliant service by entering them as input values (or connecting them to compliant services that have no input parameters).

Creating the composed service by forward chaining (i.e., starting with the first activity in the process instead of the last one) is planned but, not implemented in their prototype.

In addition to filtering on the compliance of the services in terms of their inputs and outputs, the user can apply further filtering based on the nonfunctional properties of the services. This only works for services that adhere to a specific OWL-S 'service profile' (i.e., they implement the service profile). Once the user has selected a service profile, the system renders an UI element which allows him or her to provide values for the nonfunctional properties that are specified for the selected service profile. The user can, then, apply the filter, thereby further restricting the set of services that are presented for the current composition step.

Additional to its composing functionality, Web Service Composer can also execute the composed services: The services that can be selected must be specified in OWL-S and a grounding for WSDL must be provided. Therefore, the tool can invoke the services in the composition and pass the data between the services according to the user-specified control flow.

3.2.2 Composition Analysis Tool

Kim, Spraragen and Gil introduced CAT (Composition Analysis Tool) [34], a tool which illustrates their approach to interactive workflow composition.

The focus of their work is to assist the user in the creation of computational workflows. The authors' work is not directly related to service composition. However, we can conceive a computational workflow as a service composition. The activities of the workflow compare to service operations that realize data transformations.

The authors have developed their own knowledge base format, which they use to semantically describe the components that can be used in a workflow and their input and output parameters: 'Component ontologies' describe hierarchies of components, from abstract-level components to executable components. An abstract component represents a common set of features that applies to all components of that type. 'Domain ontologies' semantically specify the data types which can serve as inputs and outputs of the components described in the component ontologies.

In CAT, the user can add components to the composition at any time. There is no need for the user to follow either a strict backward or forward chaining composition. The 'end result' of the composition can be specified by declaring outputs produced by components as the end result (or as a part of it). Control flow in CAT is described by explicitly linking inputs and outputs of different services together. Values of input parameters can also be default values from the respective ontologies or values entered by the user.

Instead of filtering the set of services that can be included in a composition, CAT provides a list of errors and warnings that point out problems of the composition at

any point in the modeling process. More than that, CAT translates these warnings into suggestions on what to do next. The idea is that consequently applying suggestions will produce a 'well-formed' workflow as a result. The authors, therefore, introduce a set of properties that must be satisfied by the composition in order to be well-formed. These properties ensure that all components' inputs are satisfied and that all components produce outputs which serve as inputs for downstream services. The latter ensures that the composition does not contain redundant activities.

Depending on whether these properties are satisfied or not, the ErrorScan algorithm (which is also provided in [34]) determines which suggestions are presented to the user. Possible suggestions include adding and removing components from the compositions or asking the user to put in values for input parameters of components that are not produced by other components in the workflow.

CAT uses heuristics to determine the ordering of the suggestions, so that more recent and more severe errors are displayed before warnings that do not necessarily have to be resolved in order for the workflow to be well-formed. It is noteworthy that the suggestions in CAT have the property of being corrective or additive: Applying a suggestion never causes more errors than it resolves.

3.2.3 PASSAT

Myers et al. present PASSAT (Plan-Authoring System based on Sketches, Advice, and Templates) [48], an interactive tool for constructing plans. PASSAT is not directly concerned with the creation of composed services, but its concepts can be mapped into the context of service composition.

PASSAT is based on hierarchical task networks (HTN) [63], while the model has been extended to realize some concepts that are outlined below. In HTN planning, a task network is a set of tasks (or service calls) that have to be carried out, as well as constraints on the ordering of these tasks. Moreover, it consists of a set of constraints that must be valid before the execution of the tasks and information about how the tasks instantiate variables. Since the variables (partly) describe the state of the world before and after the execution of a specific task, the constraints on these variables can be used to express preconditions and effects.

The HTN based approach, naturally, imposes top-down plan refinement as the planning strategy the user must adhere to: The user can start by adding tasks to a plan and refine them by applying matching HTN templates. A template consists of a set of subtasks that replace the task being refined, as well as the preconditions and effects of applying individual tasks and the entire template. It is noteworthy that the user has the possibility to override unmatched constraints when applying a template. This is especially desirable when comprehensive domain knowledge, particularly a collection of templates, cannot be provided. Task refinement is repeated until the plan contains no activities that can be further expanded.

A core feature of PASSAT is its automated planning mode, which allows the user to have the system expand all remaining tasks, applying the templates that are currently available to the system.

PASSAT also features an 'advice' mechanism that allows the user to specify high-level policies for the overall plan being created. These policies are global constraints that restrict the set of actions that the user can undertake when developing a plan. However, they can be relaxed and overridden and need not necessarily be satisfied to reach the overall goal. The automated planning mode also takes these policies into account when it selects the templates for refining the open tasks.

Opposing the strict top-down refinement approach implied by the use of HTN networks, PASSAT provides a 'plan sketch facility': This allows the user to freely arrange tasks that do not necessarily have to be fully specified and that can reside on different layers of abstraction (regarding the template hierarchy). After the user has outlined a plan sketch, the system tries to find possible expansions by applying matching templates. The user can, then, choose one of these expansions to be included in the plan and return to the normal planning mode.

PASSAT also informs the user about open tasks and outstanding information requirements in order for the plan to be completed. Therefore, it presents the user with an agenda of actions such as 'expand task', 'instantiate variable' and 'resolve constraint'.

The system helps the user to choose from the applicable templates at a given composition step by keeping track of past user experience: A statistic about how often a template has been applied in plan refinement is encoded in the templates.

3.2.4 IRS-III

Hakimpour et al. introduced Internet Reasoning Service (IRS) [27], a Semantic Web Services framework. One of their implementations, IRS-III, includes a tool that supports a user-guided interactive composition approach by recommending component Web services according to the composition context.

Their approach uses Web Services Modeling Ontology (WSMO) [54] as the language to semantically describe the functionality of the Web services. In IRS-III, Web services are represented by WSMO 'goals':

WSMO introduces the concept of goals to represent the objectives of users when consulting a Web service. A goal is a subset of a Web service's capability that is of particular interest for the user, namely the service's outputs and effects. According to Hutter, this reflects the so-called goal driven approach in AI planning [31].

Similar to Web Service Composer, the user starts with adding the goal (i.e., a Web service) that produces the desired end result of the composition. The first goal can either be selected from a list containing all goals, or by searching for an appropriate goal. The inputs of this goal must, then, be fed by other goals or values entered by the user. As in Web Service Composer, the available goals at each composition step are filtered: Only these goals that produce outputs that deliver the desired input for the downstream goal can be selected.

The tool also features the execution of the composed services. During the execution, the orchestration engine queries the user to provide values for the inputs that have not been assigned a goal or a value at design time.

In IRS-III, it is possible to introduce WSMO goal-to-goal mediators into the composition. This is necessary, when two goals are to be connected that have been specified by different parties. In such cases, it cannot be ensured that the same ontologies and, thus, the same semantic descriptions for the inputs and outputs were used by the different parties. However, if two types in different ontologies describe the same concept, the user can specify a mapping between them in a mediator.

The tool also allows if-then-else control operators to be added to the service composition.

3.2.5 SSDC

The last one of the approaches to be presented for semi-automated composition was developed by SAP Research in collaboration with Carnegy Mellon University. Rao, Dimitrov, Hofmann and Sadeh propose Smart Service Discovery and Composition (SSDC) [53], an extension of SAP's Guided Procedures that allows the modeler to specify a set of typed variables that should be produced by a Guided Procedure[2]. Their tool, then, builds a sequence out of a set of blocks, which have been annotated with metadata to link their input and output data types to an ontology. While the authors refer to their extension of Guided Procedures as a semi-automated approach, the described mechanism is, in fact, a typical example for automatic composition, except that the modeler has a user interface for specifying the 'goal' of the composition. While it was initially planned to continue the project in order to develop a true semi-automated approach, the project was discontinued.

[2]See section 2.2.1.1 for a short overview of Guided Procedures

4. A Motivating Scenario for Semi-Automated Composition

As SAP's Business Process Platform (BPP) is still under development, there are, currently, no core business processes which are fully supported in the form of end-to-end enterprise service compositions. However, the scenario used to motivate the functionality proposed in this thesis should be as close as possible to the final product. It is to build upon a set of services that resemble the enterprise services to be shipped with BPP in order to ensure its practical relevance. The scenario is taken from a recent SAP product: Duet[1], which is jointly developed by Microsoft and SAP, is the first application which completely builds upon enterprise services. Duet is an extension of Microsoft Outlook which provides integration with mySAP ERP. In doing so, a small number of day-to-day activities conducted by business users can be carried out completely inside Outlook. Duet comprises four sub-applications, namely 'Budget Monitoring', 'Time Management', 'Leave Management', and 'Organization Management'. The main benefit is that the user has no longer to manually synchronize the data in Outlook E-Mails and calendar items with transactions in the mySAP ERP system. SAP has developed enterprise services to cover functionality provided by the four Duet applications. While having been developed for Duet, the services will later be reused for applications running on top of BPP.

The process flow among the enterprise services used in Duet is not explicitly separated from the application. To obtain a business process that is composed of enterprise services, we analyzed a sub-application of Duet, namely the 'Leave Management' application. The typical actions carried out by the user in Outlook when filing a leave request trigger certain enterprise service calls. Those have been extracted and composed into a business process 'leave request' which will be described in section 4.1. Formal semantic descriptions, including an ontology describing the leave request domain will be developed in section 4.2.

[1] See `http://www.duet.com`

4.1 Scenario Overview

The leave request scenario consists of two parts: First, an employee files a leave request. Second, his manager approves or denies that leave request. Therefore, the two roles, 'employee' and 'manager', participate in the leave request process. The complete scenario is depicted in figure 4.1 using the Business Process Modeling Notation (BPMN) [69]. The tasks in the diagram correspond to enterprise service operations. To describe what the services do, we extend the BPMN syntax so that we can visualize WSMO service capabilities [54]: The inputs that a service consumes and the conditions regarding these inputs make up the 'precondition'. The outputs of a service and the relation between input and output constitute its 'postcondition'. Dependencies on the real world, such as constraints that must be valid at the time a service is executed, are encoded in the 'assumption'. The changes in the real world as a result of the execution of a service are encoded in the 'effect'.

In the following, we will first describe the service operations which are invoked on behalf of the employee; afterwards we will discuss the service operations used for the manager role.

4.1.1 Employee services

Before an employee files a leave request, he or she will, typically, try to get an overview of his time balance and suitable dates for the leave. Duet will collect this information when the leave request application pane is opened in Microsoft Outlook. Therefore, Duet will call the following four enterprise service operations in parallel:

- **Read Employee Time Account** This operation returns the current time balance that is stored for a specific employee in the ERP system. The time balance is composed of three time accounts, one for paid vacation, one for overtime and one for sick leave. The current balance for each time account is returned together with the amount of days that the employee is entitled for. The operation takes an employee object and a key date as input. The employee object uniquely refers to the employee on whose behalf the operation is invoked. The key date represents the date for which the balances are returned. In the case of Duet, this key date is the current date, i.e. the date of the day on which the operation is called.

- **Read Leave Request Configuration by Employee** This operation outputs the leave configuration for a specific employee as stored in the ERP system. The leave configuration includes the types of leaves that this employee can request, such as a full day leave, a half day leave, or sick leave. The operation takes an employee object and a key date as input, exactly as *Read Employee Time Account* does.

- **Find Leave Request By Employee** It may be the case that an employee has recently filed other leave requests which have not been processed, yet. This operation returns an employee's pending leave requests so that he or she can consider them together with the time balance. The operation takes an employee object as input.

Figure 4.1: Leave Request Scenario

- **Find Leave Request Allowed Approver by Employee** A leave request must be approved by the line manager of the employee filing it. In some cases, a different approver or no approval is required, for example, when a person can authorize his or her own leave. This operation returns the employee object corresponding to the allowed approver for the leave request. It takes an employee object as input.

The information retrieved by the four service operations described above is visualized in Duet and the employee can decide on a day or a period that is suitable for the planned leave browsing his Outlook calendar. When a date or a period has been chosen, the employee can file the leave request. This yields the sequential invocation of the two following operations:

- **Check Create Leave Request** Before a leave request is created in the ERP system, it has to be checked for plausibility. This operation takes the same inputs as the operation that, actually, creates the leave request. Besides the employee object the operation takes the leave period and the leave type as inputs. The leave type can be either a full day leave, a half day leave, illness with a certificate from a doctor or illness without certificate. When this operation is invoked, a leave request is created in the ERP system, followed by an immediate roll-back. If the request can be created successfully, the plausibility check returns a positive result. Otherwise, the result is negative.
- **Employee Leave Request** After the plausibility check has been successful, this operation, finally, creates the leave request in the ERP system. Inputs are an employee object, leave period and leave type. As a result, a leave request is created.

4.1.2 Manager services

At this point, the leave request process is completed from the employee's point of view. If the newly created leave request requires approval, the process continues from the perspective of the approving manager. Whether or not approval is required essentially depends on the type of leave: Leaves which are due to illness are not to be approved, they are only entered into the system.

The manager receives an e-mail when a leave request is pending to be approved by him or her. When the manager opens the respective Duet action pane, the following operation is called to retrieve information to be visualized in Duet about new leave requests from the ERP system:

- **Find Leave Request by ID** This operation fetches a leave request from the ERP system when it is called. It takes a leave request identifyer as input and returns the complete leave request business object, which contains all the details of the leave request.

To support his or her decision on whether or not to approve an employee's leave request, the manager can chose to get an overview of the absences of the employee's in the team that he or she supervises. If a team overview is requested, the following three operations are sequentially invoked, otherwise, they are skipped:

- **Find Reporting Employee by Employee** This operation returns the list of employees who are reporting to a manager. It takes a manger object as input and returns a list of employee business objects.
- **Read Employee Time Account** This operation was already described above. Here, the operation is used to obtain the time balances for all employees on the team and is therefore repeated for each employee.
- **Find Leave Request By Employee** Similarly, this operation was already discussed. The operation is used to retrieve pending leave requests from all team members. The manager needs to be aware of pending requests, because it can happen that other team members plan leaves for the same date as the leave request under review, does.

Having reviewed the absence situation of the team, the manager can, then, take the decision on whether to approve or reject the leave request. If the manager decides to approve the leave request, the invocation of the two following services will be yielded:

- **Check Approve Leave Request** Before a leave request is set to an 'approved' status in the ERP system, a plausibility check is conducted (similar to the creation of the leave request). This operation takes a leave request object as input. When this operation is invoked, the leave request is approved in the ERP system, followed by an immediate roll-back. If the approval succeeded, the plausibility check returns a positive result. Otherwise, the plausibility check fails.
- **Approve Leave Request** After the plausibility check has been successful, this operation, finally, sets the leave request to 'approved' in the ERP system. It takes a leave request object as input, exactly as the preceding plausibility check does.

The operations **Check Reject Leave Request** and **Reject Leave Request** function in the same way as the operations to approve the leave request do, except that the goal is to reject the request.

4.2 Scenario Specification

Before (semi-)automated composition techniques can be applied, semantic service descriptions must be available for the services from which compositions are to be created. Before we illustrate typical semi-automated composition functionality in

chapter 5 with the scenario services presented above, we will first develop semantic descriptions for them throughout the remainder of this chapter. We will start with formalizing the leave request domain in the form of an ontology. Afterwards, we will semantically describe the service operations used in the presented scenario.

Based on the evaluation of languages for semantic Web service specification, presented in section 3.1.6, we chose WSMO [54] for semantically describing the domain and the service operations.

4.2.1 Ontology

In order to allow reasoning in a specific domain, an ontology for that domain must be available. We have encoded the leave request domain in a WSMO ontology. The full ontology in WSML [16] is listed in appendix A.1. In the following, the key ideas that have been applied to develop the ontology will be briefly described.

4.2.1.1 Concepts

The concepts in the ontology are derived from the SAP Global Data Types (GDTs) [58] which are used for the parameters of the scenario services. However, we did not create an ontology for the complete GDT catalog, because this is not within the scope of this work. Instead, the presented scenario ontology contains only concepts which refer to GDTs that are used in the leave request scenario. The concepts that are part of the created ontology are depicted in figure 4.2, visualizing the subclassing among the various concepts.

In the process flow depicted in figure 4.1, the two roles, 'employee' and 'manager', are introduced. These roles are also represented as concepts in the ontology. More specifically, the ontology contains a concept `Person`, which has a subconcept `Employee`, which itself has a subconcept `Manager`. This is realized using the *subConceptOf* operator in WSML, as depicted in listing 4.1. The concepts `Employee` and `Manager` correspond to the business objects 'employeee' and 'manager' as specified in the GDT catalog.

```
1 concept Person
    FirstName ofType _string
3   LastName ofType _string

5 concept Employee subConceptOf Person
    EmployeeID ofType (1 1) _integer
7
  concept Manager subConceptOf Employee
```

Listing 4.1: The concept 'person' and its subconcepts

The ontology contains a number of enumeration types, e.g. `LeaveRequestState`, `TimeAccountType` and `Result`. The chosen language dialect of WSML for the specification of the ontology, WSML-Flight, does, however, not support enumeration types. The possible values of the enumeration types have, therefore, been realized as subconcepts of the enumeration types in the ontology.

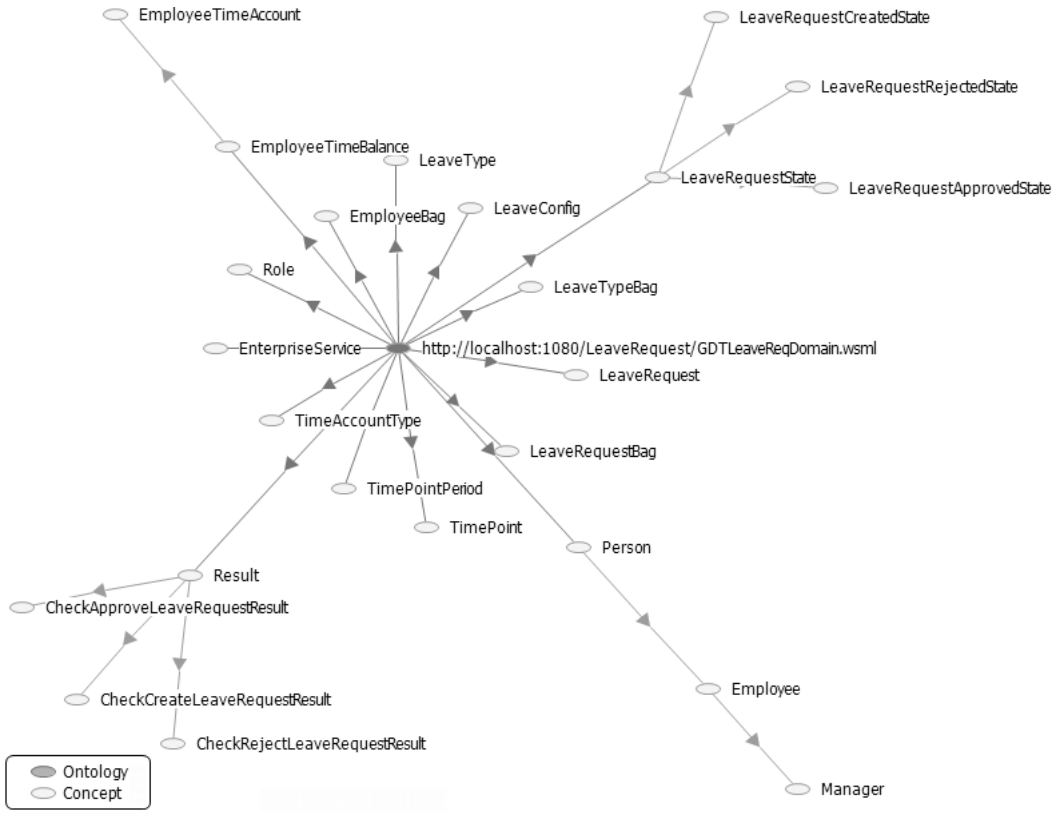

Figure 4.2: Concepts in the leave request domain ontology

WSML-Flight does also not provide for a list type. However, some of the service operations consume or produce lists of a specific data types when the are invoked, as opposed to producing a single value of a type. For the concepts that would usually require a list type, we have created so-called 'bag' concepts to work around this problem. A bag concept has an attribute which is typed as the contained concept with a minimum multiplicity of one.

```
concept EmployeeBag
  membersEMP ofType (1 *) Employee
```

Listing 4.2: A bag of 'employee' concepts

Listing 4.2 shows an example of a bag type. An `EmployeeBag` concept is defined which has an attribute typed as the Employee concept (see listing 4.1). This attribute can be used to store one ore more instances of the concept `Employee`.

4.2.1.2 Relations

The ontology contains the relations depicted in figure 4.3. Relations are defined among an arbitrary number of concepts in the ontology. They express constraints on concrete instances of concepts, which take part in a relation.

Listing 4.3 shows how relations are specified in WSML, using the relation `has-Requestor` as an example: An instance of this relation assigns an instance of the concept `Employee` to an instance of the concept `LeaveRequest`.

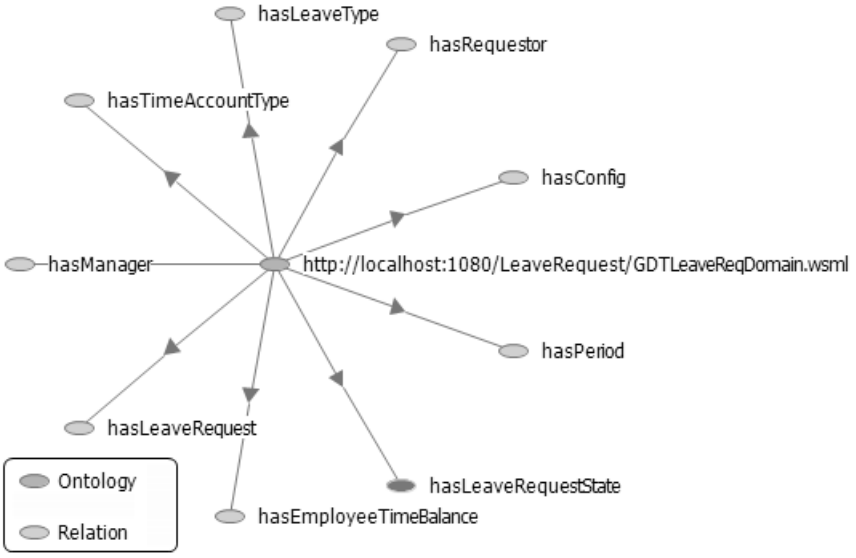

Figure 4.3: Relations in the leave request domain ontology

```
relation hasRequestor(ofType LeaveRequest, ofType Employee)
```

Listing 4.3: The relation 'hasRequestor'

4.2.2 Service Operations

While the capabilities of the service operations used in the leave request scenario were already described in natural language in section 4.1, a formal description of the capabilities is a necessary prerequisite for semi-automated service composition. In the following, we will describe the main ideas that have been employed to create the semantic service specifications in WSML using examples. The complete service descriptions in WSML are listed in appendix A.2. In WSMO, a capability of a Web service may contain a precondition, a postcondition, an assumption, an effect as well as nonfunctional properties and shared variables. These elements are discussed briefly using the leave request scenario services as an example.

4.2.2.1 Preconditions and Assumptions

In WSMO, preconditions and assumptions express the state that has to be satisfied so that the described operation can be invoked. The precondition lists the information artifacts, i.e. the input parameters, that must be available at the point in time when the described service is to be invoked. These information artifacts are described using the concepts that have been specified in an ontology. The assumption lists a number of conditions regarding the input parameters. This is done using the relations that have also been specified in an ontology. Listing 4.4 shows excerpts of the precondition and the assumption of the *Check Reject Leave Request* operation. The precondition states that two facts must be available for the operation to be invocable. One fact is an instance of the concept `LeaveRequest`, the other fact is an instance of the concept `LeaveRequestCreatedState`. What's more, the second fact

must have an attribute 'state' which has the value 'requestedState'. The assumption states that the two facts, described in the precondition, participate in the relation `hasLeaveRequestState`. This expresses that the state, described by the second fact, belongs exactly to the leave request described by the first fact.

```
precondition
  definedBy
    ?lrq memberOf dO#LeaveRequest
      and ?state memberOf LeaveRequestCreatedState
      and ?state[state hasValue 'requestedState'].
assumption
  definedBy
    dO#hasLeaveRequestState(?lrq, ?state).
```

Listing 4.4: Precondition and assumption of the 'Check Reject Leave Request' operation

4.2.2.2 Postconditions and Effects

Postconditions and effects are used to model the point in time after a service has been invoked. More precisely, they describe the information state and the state of the world after the invocation. Listing 4.5 shows an excerpt of the postcondition and the effect of the *Find Leave Request Allowed Approver by Employee* operation. The service returns the line manager of a given employee. Therefore, the postcondition states that the service produces a fact that is an instance of the concept `Manager`. The effect states that the person corresponding to the produced fact is, in the real world, the line manager of the person described by the fact that the service consumes, which is not shown in the listing.

```
postcondition
  definedBy
    ?mgr memberOf dO#Manager
effect
  definedBy
    dO#hasManager(?emp,?mgr).
```

Listing 4.5: Postcondition and effect of the 'Find Leave Request Allowed Approver by Employee' operation

4.2.2.3 Shared Variables

In order to use the same variables throughout multiple segments of a WSMO capability, the respective identifiers have to be added to the list of shared variables. In the capability of the *Find Leave Request Allowed Approver by Employee* operation, for example, the variable `?mgr` is used in both the precondition and the effect and, actually, refers to the same fact in both occurences. The variable `?mgr` must, therefore, be declared as a shared variable. The precondition of *Find Leave Request Allowed Approver by Employee* is not shown in listing 4.5. However, the precondition contains a fact `?emp` which is also referred to in the effect of the capability and must, consequently, also be added to the list of shared variables. This is shown in listing 4.6.

```
sharedVariables {?emp, ?mgr}
```

Listing 4.6: Declaration of a shared variable

4.2.2.4 Nonfunctional Properties

Noncuntional properties model characteristics of a service operation that are independent of the data transformation or the change in the state of the world that a service realizes. Nonfunctional properties are, for example, used in the semantic descriptions of our scenario services to model the operations' affilliations with enterprise services containing the operations. An example stating that the *Find Leave Request Allowed Approver by Employee* operation belongs to the 'Time & Leave Management' enterprise service is shown in listing 4.7. The nonfunctional property `EnterpriseService` is realized as a concept in the ontology.

```
1 nfp
    dO#EnterpriseService hasValue 'Time & Leave Management'
3 endnfp
```

Listing 4.7: A nonfunctional property of a service operation

5. Mixed Initiative Features for Semi-Automated Composition

In this chapter, we will describe three mixed initiative features which are characteristic for semi-automated service composition. As depicted in figure 5.1, these features can be seen as use cases for semi-automated composition environments that help them to overcome the problems of complexity, inflexibility and error-proneness akin to conventional service compositions, as described in chapter 1.

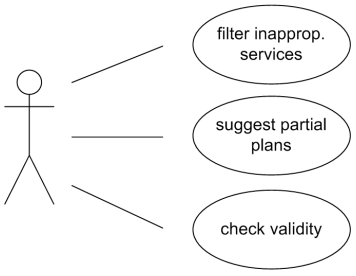

Figure 5.1: Mixed initiative features

To showcase how semi-automated composition techniques can ease the modeling of a business process, we will work with the scenario that was presented in chapter 4. The business process in the Leave Request Scenario will be constructed step by step, which supports and motivates the introduction of the three mixed initiative features.

This chapter is organized as follows: First, section 5.1 will refresh our understanding of how to semantically describe the capabilities of a service. We start with the definition of a formal model for service compositions on which we build the formal description of service capabilities used throughout this thesis. The mixed initiative features will also build upon this formal model, when they are defined throughout the remainder of this chapter. Each of the three features will first be motivated with our business scenario from chapter 4. Then, each of the three mixed initiative features will be described in detail.

5.1 A Formal Model for Service Compositions and Capabilities

As already stated in section 1.1.1, services can have capabilities that are *information-providing* or *world-altering* (or both). The invocation of information-providing services results in a change in the information state at a given point in time. In WSMO terminology [54], the data transformation accomplished by a Web service operation is described by the `precondition` and `postcondition` of an operation. Another class of service capabilities are the so-called world-altering capabilities. As opposed to information-providing capabilities, world-altering capabilities have an irreversible `effect` on the state of the world. An example would be a payment service that has the effect of charging a credit card when it is invoked. Similarly, there might be certain `assumptions` about the state of the world that must be verified, before a service can run. Reconsidering the leave request example, such an assumption could be that the person trying to execute the operation that approves a leave request is the line manager of the employee requesting the leave and is, therefore, authorized to approve the request. Throughout the remainder of this thesis, we will use the terms precondition, postcondition, assumption and effect to describe service capabilities. We will, first, give a formal definition of the terms 'service composition' and 'ontology' as the foundation for the definitions necessary to describe information-providing and world-altering services.

5.1.1 Service Compositions and Ontologies

Usually, we regard business processes (or service compositions, respectively) as a graph of activities (or service operations). These activities are in a partial order that denotes the execution flow of the process. However, we can also look at a service composition as a bipartite graph of states and transitions. In doing so, the transitions denote the service operations. Before and after each transition there is a state. We, then, think of the invocation of a service as a transition from one state A to another state B. States capture the informational state and the state of the world at a given point in a service composition. To capture the essential concepts of process modeling, such as alternatives and parallelism, our graph representation has to contain more than two node types, i.e. states and service operations. In the following, a formal definition of the term 'service composition' will be given:

Definition 5.1 (Service composition) We define a service composition as a k-partite graph

$$G := < V, E >$$

with $k = 6$, $V = S \cup OP \cup AS \cup AJ \cup OS \cup OJ \cup LB \cup LE$.

The set of vertices V is partitioned as follows:

- S is the set of states
- OP is the set of service operations
- AS is the set of AND-splits

- AJ is the set of AND-joins
- OS is the set of OR-splits
- OJ is the set of OR-joins
- LB is the set of nodes representing the beginning of a loop
- LE is the set of nodes representing the end of a loop

Remark The distinction between inclusive and exclusive OR-semantics is not necessary for the introduction of the mixed initiative features for semi-automated service composition. The definition of a service composition is, therefore, limited to a notion of OR-splits and OR-joins covering both OR facettes.

Remark To denote that an edge e connects two vertices v_1 and v_2, we write

$$v_1 \xrightarrow{e} v_2.$$

Remark If an edge e connects two vertices v_1 and v_2, then either v_1 or v_2 is a state.

$$\forall e \in E \text{ with } v_1 \xrightarrow{e} v_2 : v_1 \in S \oplus v_2 \in S$$

Remark A state can have a maximum number of one incoming and one outgoing edge:

$$|\{e : v \xrightarrow{e} s\}| \leq 1,\ v \in V,\ s \in S,\ \forall\ e \in E$$
$$|\{e : s \xrightarrow{e} v\}| \leq 1,\ v \in V,\ s \in S,\ \forall\ e \in E$$

Remark For every AND-split node, there is a corresponding AND-join node in the service composition. The same holds true for OR-splits and OR-joins as well as for nodes representing the beginning of a loop and nodes representing the end of a loop.

$$|AS| = |AJ|$$
$$|OS| = |OJ|$$
$$|LE| = |LB|$$

Remark The first vertex in a service composition is a state. It is labeled s_0:

$$\exists s_0 \in S \text{ such that } \nexists v \in V \text{ with } v \xrightarrow{e} s_0,\ e \in E$$

Remark The last vertex in a service composition is a state:

$$\exists s \in S \text{ such that } \nexists v \in V \text{ with } s \xrightarrow{e} v,\ e \in E$$

Example Figure 5.2 depicts a sample service composition based on the formal model defined in definition 5.1. All vertices and edges are annotated with the subset of V or E they belong to.

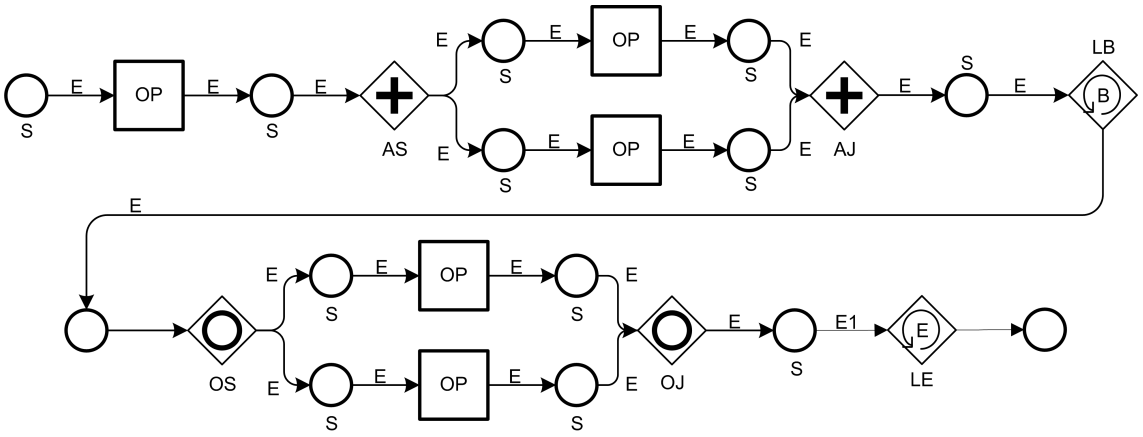

Figure 5.2: A sample service composition graph according to definition 5.1.

Definition 5.2 (Ontology) An ontology Ω consists of concepts, attributes of the concepts, and relations between the concepts. The set $C \subseteq \Omega$ contains all the concepts c_i that are specified in Ω. An ontology can also contain an arbitrary number of n-ary relations between the concepts. The set $R \subseteq \Omega$ contains all the relations $r_i \subseteq C \times \cdots \times C$ that are specified in Ω.

Remark Other common definitions of the term 'ontology' [54, 42] also include the notion of 'individuals'. Individuals are instances of a concept that are specified directly in an ontology. In our model, however, individuals as part of the ontology are not necessary. They are, therefore, excluded from our definition of an ontology. However, there are instances of concepts that are not pre-defined in an ontology. These are called facts (see defnition 5.3).

Remark A concept c_2 can be derived from another concept c_1. In doing so, a hierarchy of the concepts in the ontology is built. The function $super : C \rightarrow C$ returns the superconcept of a given concept, i.e. the concept it is derived from. In contrast, the function $sub : C \rightarrow \mathcal{P}(C)$ returns the set of subconcepts of a given concept. $\mathcal{P}(C)$ is the power set of C.

5.1.2 The Class of Information-Providing Services

Based on the above definitions, we will, now, introduce the concepts necessary to enhance our model to capture the essence of information-providing services.

Definition 5.3 (Fact) A fact $f \in F$ is an instance of a concept in Ω. The set F contains all facts that exist in an environment. The function $concept : F \rightarrow C$ returns the concept in Ω that a given fact is an instance of. There can be multiple facts corresponding to the same concept in Ω.

Definition 5.4 (Constraint) A constraint $co \in R_{co}$ is an instance of a relation in Ω. A constraint $co_i \in R_{co}$ is an n-ary relation between facts. The set R_{co} contains all the constraints $co_i \subseteq F \times \cdots \times F$ that exist in an environment. The function $relation : R_{co} \rightarrow R$ returns the relation in Ω that a given constraint is an instance of. There can be multiple constraints corresponding to the same relation in Ω.

Definition 5.5 (Precondition) A precondition is the set of facts that a service operation consumes when it is invoked. The function $pre : OP \rightarrow \mathcal{P}(F)$ returns the set of facts that are contained in the precondition of a service operation. $\mathcal{P}(F)$ is the power set of F.

Definition 5.6 (Postcondition) A postcondition is the set of facts that a service operation produces when it is invoked. The function $post : OP \rightarrow \mathcal{P}(F)$ returns the set of facts that are contained in the postcondition of a service operation. $\mathcal{P}(F)$ is the power set of F.

Definition 5.7 (Information state) The information state $inf(s)$ is the partition of a state s containing the set of all postconditions of all service operations preceding s, as well as the facts that are available in the initial state (s_0) of the composition. The set of facts $\epsilon(s) \subseteq inf(s)$ contains the facts that are made available by the execution environment in state s. The function $inf : S \rightarrow \mathcal{P}(F)$ returns the set of facts that are contained in the information state partition of a state. It is recursively defined as follows:

$$
inf(s_i) = \begin{cases}
\epsilon(s_i) \cup post(v) \cup inf(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \text{ with } s_{i-1} \xrightarrow{d} v \\
\quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in OP \\
\\
\epsilon(s_i) \cup inf(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \text{ with } s_{i-1} \xrightarrow{d} v \\
\quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in AS \\
\\
\epsilon(s_i) \cup (\bigcup inf(s_k)), \forall k \text{ such that } \exists d \in E \text{ with } s_k \xrightarrow{d} v \\
\quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in AJ \\
\\
\epsilon(s_i) \cup inf(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \text{ with } s_{i-1} \xrightarrow{d} v \\
\quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in OS \\
\\
\epsilon(s_i) \cup (\bigcap inf(s_k)), \forall k \text{ such that } \exists d \in E \text{ with } s_k \xrightarrow{d} v \\
\quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in OJ \\
\\
\epsilon(s_i) \cup inf(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \text{ with } s_{i-1} \xrightarrow{d} v \\
\quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in LB \\
\\
\epsilon(s_i) \cup inf(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \text{ with } s_{i-1} \xrightarrow{d} v \\
\quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in LE \\
\\
\epsilon(s_0) \text{ if } s_i = s_0
\end{cases}
$$

Remark The postconditions of all service operations between an AND-split and an AND-join node are combined and are part of the information state partition of the state after the AND-join node.

Remark The postconditions of all service operations between an OR-split and an OR-join node are not combined. This is due to the fact that the decision which OR-paths are executed (and, thus, which facts are actually available in the state after the OR-join) is deferred to runtime. The information state partition of the state after an OR-join is, therefore, reduced to a common denominator: The OR-path-wise intersection of the facts produced as postconditions of the service operations.

Remark In our model, a loop does not affect the information state partition of a state. During its first iteration, the information state partitions of the states between the nodes representing the beginning and the end of the loop are the same as if the loop was not present. From the second iteration on, it might be the case (for loops spanning more than one service operation) that facts from postconditions in the last iteration(s) have been produced. However, it is a runtime decision whether a loop is iterated or not. Therefore, we neglect the influence of loops on the information state, because our focus lies on design-time.

5.1.3 The Class of World-Altering Services

By elaborating the necessary definitions for the introduction of the information state partition of a given state, we have covered the set of information-providing services. In the following, we will extend our model to deal with services providing world-altering capabilities.

Definition 5.8 (Assumption) An assumption is the set of constraints that must be satisfied as a prerequisite for invoking a service operation. The function $assump : OP \rightarrow \mathcal{P}(R_{co})$ returns the set of constraints that are contained in the assumption of a service operation. $\mathcal{P}(R_{co})$ is the power set of R_{co}.

Definition 5.9 (Effect) An effect is the set of constraints that is produced by the invocation of a service operation. All constraints in this set are satisfied at the point in time after a service operation is invoked. The function $eff : OP \rightarrow \mathcal{P}(R_{co})$ returns the set of constraints that are contained in the effect of a service operation. $\mathcal{P}(R_{co})$ is the power set of R_{co}.

Definition 5.10 (World state) The world state $world(s)$ is the partition of a state s containing the set of all effects of all service operations preceding s, as well as the constraints that are defined in the initial state (s_0) of the composition. The set of constraints $\gamma(s) \subseteq world(s)$ contains the constraints that are made available by the execution environment in state s. The function $world : S \rightarrow \mathcal{P}(R_{co})$ returns the set of constraints that are contained in the world state partition of a state. It is recursively defined as follows (analogous to definition 5.7):

$$world(s_i) = \begin{cases} \gamma(s_i) \cup eff(v) \cup world(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \\ \quad \text{with } s_{i-1} \xrightarrow{d} v \text{ if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in OP \\ \\ \gamma(s_i) \cup world(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \text{ with } s_{i-1} \xrightarrow{d} v \\ \quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in AS \\ \\ \gamma(s_i) \cup (\bigcup world(s_k)), \forall k \text{ such that } \exists d \in E \text{ with } s_k \xrightarrow{d} v \\ \quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in AJ \\ \\ \gamma(s_i) \cup world(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \text{ with } s_{i-1} \xrightarrow{d} v \\ \quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in OS \\ \\ \gamma(s_i) \cup (\bigcap world(s_k)), \forall k \text{ such that } \exists d \in E \text{ with } s_k \xrightarrow{d} v \\ \quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in OJ \\ \\ \gamma(s_i) \cup world(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \text{ with } s_{i-1} \xrightarrow{d} v \\ \quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in LB \\ \\ \gamma(s_i) \cup world(s_{i-1}), \text{ with } s_{i-1} \text{ such that } \exists d \in E \text{ with } s_{i-1} \xrightarrow{d} v \\ \quad \text{if } \exists e \in E \text{ such that } v \xrightarrow{e} s_i,\ v \in LE \\ \\ \gamma(s_0) \text{ if } s_i = s_0 \end{cases}$$

Remark For the definition of the world state partition of a state, AND-splits, AND-joins, OR-splits, OR-joins, nodes representing the beginning of a loop and nodes representing the end of a loop are treated in the same way as in definition 5.7.

5.2 Filter Inappropriate Services

The first mixed initiative that will be discussed is named 'Filter Inappropriate Services'. This feature addresses a major problem in creating composed services: The number of activities that are available as building blocks for the composition might be extremely high, depending on the domain. In the context of SAP, for example, the Enterprise Service Repository, the central repository of enterprise services, contains more than 12,000 core services and more than 240 compound services to date[1]. Several hundred more are. currently, being examined for approval by the Process Integration Council (PIC) at SAP. Another several hundred service operations for industry specific solutions have not even been submitted for approval. The high

[1]Source: X8R, Software Component SAP BASIS 7.10 (as of Nov 1, 2006)

number of available service operations results from the fact that each business object provides at least 10 service operations, which is due to the interface patterns described in section 2.2.3. This leads to a complexity that is unmanageable for a modeler of service compositions. Particularly when executable services are to be realized with service compositions that are created by users with a non-technical background, as envisioned in SAP's long-term strategy, a modeling tool for creating service compositions should support the user as far as possible. For the user cannot oversee such a vast amount of available options as presented in the enterprise services example, it is desirable to filter the set of available services. Such filtering can be done based on semantic service descriptions.

5.2.1 Business Scenario

In order to illustrate the 'Filter Inappropriate Services' feature as well as the other mixed initiative features, we will use the leave request scenario presented in chapter 4. While the whole process, depicted in figure 4.1, could be constructed by the aid of the mixed initiative features, we limit ourselves to constructing the first part of the leave request business process, up to the point where the employee creates the leave request. This part of the process is depicted in figure 5.3.

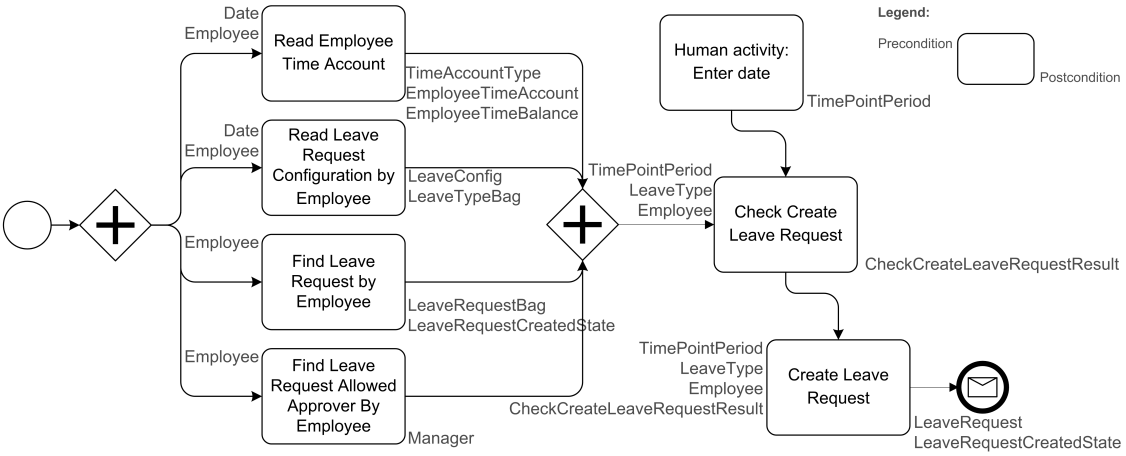

Figure 5.3: The leave request scenario from an employee perspective

We have implemented a semi-automated business process orchestration tool that realizes the three mixed initiative features. Our implementation serves as an example for a semi-automated composition environment. The presented business scenario outlines the activities that a modeler would carry out in this modeling environment, which is in this case provided by our tool, to build the leave request process. The realization of the mixed initiative features and the implementation of our tool will be discussed in chapters 7 and 8.

When the leave request is to be created from scratch, the tool will first retrieve all available Web services. The modeler starts by adding the role 'employee' to the composition by selecting this role from a list of all available roles, e.g. 'supplier', 'customer', 'manager' or 'sales representative'. Our tool, then, assumes the implicit availability of a variable of the complex type 'employee', representing the person

who takes part in the business process in this role. The tool is, now, able to filter the list of available service operations down to those that require an employee object as an input. Our experiments have shown that filtering all services that would not be invocable in the current step of the composition is too strict. The tool will, therefore, also present service operations that are nearly invocable in the sense that only one input data type is missing. Using this technique, we are able to retrieve very reasonable suggestions from SAP's service repository. The service operations are logically grouped around so-called enterprise services. In our example, the modeler would, therefore, then expand the 'Time and Leave Management' enterprise service and select the first four operations from figure 5.3. As there are no dependencies among the four activities, the user connects the operations using a parallel control flow. Figure 5.4 gives an example of what this could look like in a semi-automated modeling environment.

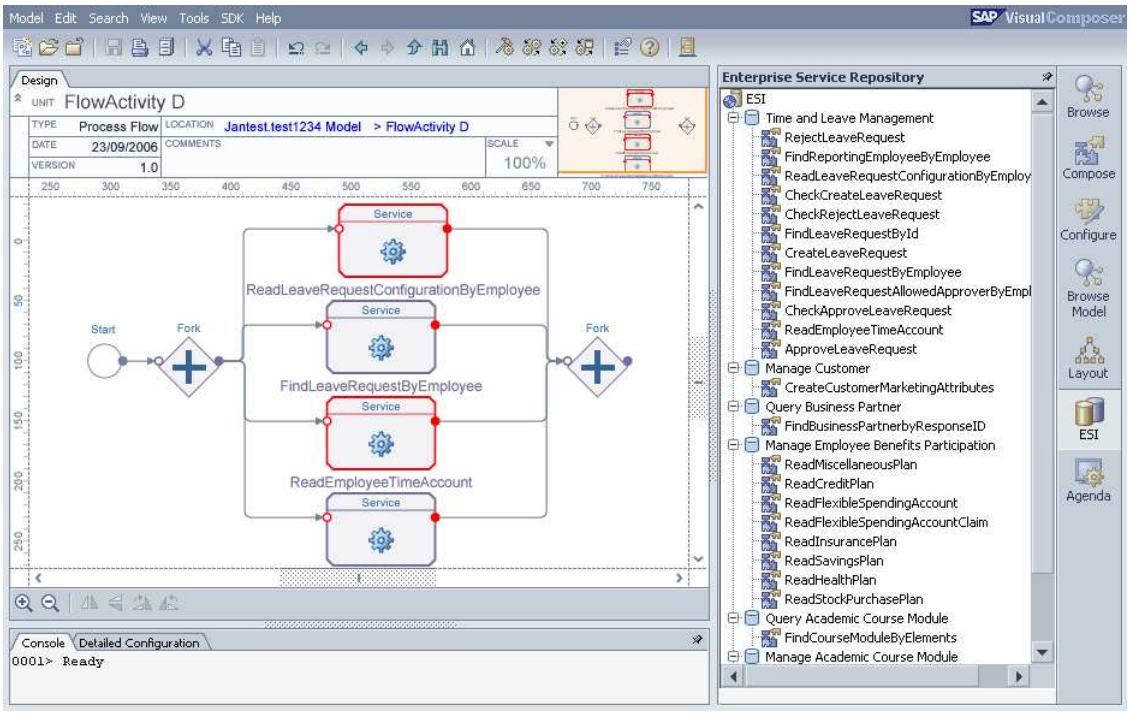

Figure 5.4: Screenshot of the modeling tool

At this point, the modeler is able to retrieve more service suggestions through the filtering mechanism by clicking on the merge node of the parallel split. Our tool will, then, present a list of service operations that are invocable or nearly invocable based on the union of all postconditions of the services which are in the composition so far. The postconditions, i.e. the output data types, of the operations are also depicted in figure 4.1. Amongst other things, our tool will suggest the operation *Check Create Leave Request* as a nearly invocable service. The modeler adds it to the composition and creates a link between the merge node and the operation.

5.2.2 Formal Description

When creating composed services, users select services and add them to the composition. In a given state, it is possible to filter the selection according to semantic descriptions: It is desirable that service operations that have preconditions and assumptions that are not satisfied in a given state will be filtered. This filtering will effectively reduce the number of choices presented to the user.

A filtered list should only contain service operations that are invocable in the current state of the composition. Definition 5.11 gives a formalization of the semantics of the term 'invocable'. It builds upon the definitions of the previous sections.

Definition 5.11 (Invocable) A service operation $o \in OP$ is invocable in state $s \in S$ if

$$pre(o) \subseteq inf(s) \cup \bigcup_{f \in inf(s)} \{\ g \in F \mid \exists c \in C \quad \text{such that } concept(f) = c \wedge concept(g) \in sub(c)\ \}$$
$$\wedge\ assump(o) \subseteq world(s).$$

Remark We write $s \models o$ ('s satisfies o') to denote that service operation o is invocable in state s.

Our experiments with the service repository of SAP show that the filtering of the available services operations on the basis of invocability in a state is often too strict. A filtering that is too strict can result in a situation where the number of invocable services is very low or contains services that have already been included in the composition. To provide more valuable service suggestions, a certain amount of fuzzyness can be added to the filter. This can be done by assuming the availability of facts that are not part of the information state partition of a state. In doing so, the filtering will result in a list of 'nearly invocable' service operations. They are only nearly invocable, because they have been discovered under the assumption that additional facts were available in a given state. Definition 5.12 formally introduces the concept of nearly invocable service operations.

Definition 5.12 (Nearly invocable) A service operation $o \in OP$ is *nearly* invocable to a degree $k \in \mathbb{N}$ in state $s \in S$ if

$$\exists f_1 \in F\ \wedge\ \exists f_2 \in F\ \wedge\ \ldots\ \wedge\ \exists f_k \in F \text{ such that}$$
$$\big(\ pre(o) \in inf(s)$$
$$\vee\ pre(o) \subseteq \bigcup_{f \in inf(s)} \{g \in F | \exists c \in C \text{ such that } concept(f) = c \wedge concept(g) \in sub(c)\}$$
$$\vee\ pre(o) \subseteq \{f_1, \ldots, f_k\}$$
$$\vee\ pre(o) \subseteq \{g \in F | \exists c \in C \text{ such that } concept(f_1) = c \wedge concept(g) \in sub(c)\}$$
$$\ldots$$
$$\vee\ pre(o) \subseteq \{g \in F | \exists c \in C \text{ such that } concept(f_k) = c \wedge concept(g) \in sub(c)\}\ \big)$$
$$\wedge\ assump(o) \subseteq world(s).$$

In our experiments, we were able to retrieve very reasonable service suggestions with $k = 1$. That means that adding only one fact to the state can significantly improve the filtering.

5.2.3 Possible Extensions

While filtering based on the services' capabilities restricts the set of presented services to those which are invocable or nearly invocable in the current state, the set can be further restricted by filtering based on the nonfunctional properties of the services operations. Nonfunctional properties do not only offer a possibility to record juridically relevant information such as a publisher's name and address, but also quality indicators for services. Such indicators can be measures that address the performance (in terms of response time), error rate or robustness of a service, as well as issues like scalability, reliability, geographical coverage, invocation cost, and many more. When creating service compositions, the user may find himself or herself in a situation where more than one available service offers the functionality that he or she looks for at a specific point in the composition, i.e. a state in the composition. At this point, the editor should allow the user to assign values to the nonfunctional properties of the filtered services. These values are, then, evaluated by the semi-automated composition environment. Only those services that both provide the desired functionality and comply with the user-specified value of the nonfunctional properties are presented for selection and remain in the filtered list. It should also be possible for the user to specify weighted combinations of values of nonfunctional properties to obtain a further filtering of the list. However, filtering on nonfunctional properties implies a strong requirement for the semantic service specifications: All the semantic descriptions of the services providing equivalent functionality must contain the same set of nonfunctional properties. Otherwise, the filtering with nonfunctional properties cannot be applied to all service operations that have the same capabilities.

Another possibility to further enhance the filtering is to introduce an ordering of the filtered list of (nearly) invocable services according to the 'quality of match' between the information state and the precondition. Li and Horrocks [41] propose a classification of the 'goodness' of a match which can be taken into account for ordering the list of service suggestions. All the individual facts of a precondition of an operation are matched against the information state partition of the current composition state. If the concepts required in a precondition are contained in the current state (i.e., an 'exact match', see definition 5.13), the operation should be given a high ranking in the list. In the second place, service operations with preconditions which are satisfied by facts representing more specific concepts than those required by the operation (i.e., a 'plugin match', see definition 5.14) should be shown. A list of plugin matches can, furthermore, be ordered according to the individual match distances between the required and the available concepts, which is also formally introduced in definition 5.14.

Definition 5.13 (Exact Match) A fact that is part of the information state partition of a state $s \in S$ exactly matches a fact that is part of the precondition of a service operation $o \in OP$ if

$$\exists f \in pre(o) \ \wedge \ \exists g \in inf(s) \text{ such that } concept(f) = concept(g).$$

Definition 5.14 (Plugin Match) A plugin match exists if a fact f, which is part of the precondition of a service operation, is matched by another fact h which is part of the information state partition of s, where h is an instance of a subconcept of the concept that f is an instance of.

$$\begin{aligned} &\exists f \in pre(o) \\ &\wedge \ \exists g \in inf(s) \text{ such that } concept(f) \neq concept(g) \\ &\wedge \ \exists h \in inf(s) \text{ such that } concept(f) = concept(h) \\ &\text{with } concept(h) \subseteq sub(concept(g)). \end{aligned}$$

Remark The match distance k denotes the number of indirections between the concepts g and h in the ontology Ω. It can be formally defined using the function *super* that was introduced in definition 5.2:

$$super^k(concept(h)) = concept(g)$$

A third possibility to enhance the Filter Inappropriate Services feature is to order the list of (nearly) invocable services according to a rating of how often the user has selected the particular services. This extension does not, primarily, build upon the use semantic service descriptions. It will, therefore, not be discussed here in greater detail. However, it would be conceivable to realize a rating facility using a nonfunctional property in the semantic service descriptions: Such an attribute could track how often the modeler selects a service operation. The editor could update these descriptions each time a service is selected, increasing the counter in the nonfunctional property.

5.3 Check Validity

Since the human planner has full control over the modeling of the business process in manual and semi-automated environments, he or she is likely to introduce errors into the composition. It is, therefore, necessary to provide the possibility to check the overall process validity. By validity we refer to the semantic validity of a process as opposed to its syntactic validity. Syntactic properties for valid process models include, for example, soundness, which is defined by van der Aalst in [66]. Syntactic validity can be verified statically, i.e. by investigating the structure of the service composition. Checking for soundness takes the dynamics of a composition into account, for example, when checking whether or not activities are 'live' or the composition can terminate in all possible states. However, when semantic descriptions for the activities of process are available, we are able to define correctness criteria for

processes on the semantic level. Corresponding validity checks that operate on such correctness criteria could be triggered by the user at the end of the modeling process. In a semi-automated composition environment, however, the semantic validation of the process should be interwoven with the actual modeling of the process. The user should be informed about problems with the composition in an unobtrusive way.

5.3.1 Business Scenario

In our business scenario, which is depicted in figure 5.3, the last step was that the modeler added the nearly invocable operation *Check Create Leave Request* to the composition. The tool highlights operations for which problems are tracked. As the added operation is not invocable, but nearly invocable, one input type is missing. The tool, therefore, marks the operation with a red border. This can also be seen in figure 5.4, where two out of four activities are highlighted. By clicking on the *Check Create Leave Request* operation, the user can open a panel showing the input and output types of the operation as inferred from the pre- and postconditions. The user can see that all input types of the operation are currently available in the composition, except *TimePointPeriod*, which is also highlighted using red color in this drill-down view. The user can also get an overview of all current problems of the composition by looking at the agenda. An example of what such an agenda could look like is depicted in figure 5.5.

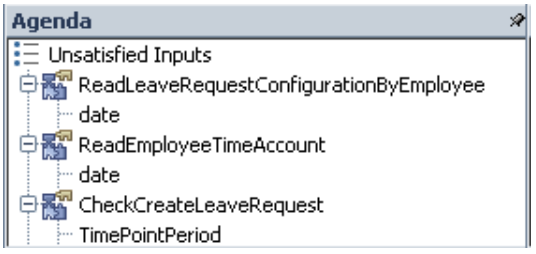

Figure 5.5: Agenda summarizing problems in a composition

The missing parameter *TimePointPeriod* represents the date or period for which the employee intends to request a leave. As our scenario has been taken from Duet, this data is provided by Microsoft Outlook after the user selects a date from the calendar. In our example, the modeler, therefore, creates a human activity that produces a *TimePointPeriod* variable as an output. The modeler connects the human activity with the *Check Create Leave Request* operation. The coloring of the operation and the *TimePointPeriod* input type in the parameter view disappears and the issue is removed from the agenda.

5.3.2 Formal Description

During the process of modeling a composed service using a semi-automated composition environment, the modeler can be supported by being provided with a list of problems. These problems reflect the semantic validity of a service composition. The semantic validity is based upon a set of correctness criteria which can be validated

at any point during the creation of a composed service. The work of Kim, Spraragen and Gil [34] has inspired our correctness criteria for composed services.

As stated in definition 5.5, the *precondition* of a service operation consists of a number of input types that this operation consumes when it is invoked. When creating a service composition, the modeler has to ensure that all input types of an operation are available before it is invoked. If a service composition contains an activitiy with unsatisfied inputs, its execution will fail at the point when this activity is to be invoked. Definition 5.15 formally introduces the notion of an unsatisfied input.

Definition 5.15 (Unsatisfied input) A fact $f \in F \subseteq \Omega$ is an unsatisfied input of service operation $o \in OP$ if

$$
\begin{aligned}
& f \in pre(o) \\
& \wedge f \notin inf(s) \\
& \wedge f \notin \bigcup_{g \in inf(s)} \{h \in F | \exists c \in C \text{ such that } concept(g) = c \wedge concept(h) \in sub(c)\}
\end{aligned}
$$

with $s \in S \wedge e \in E$ such that $s \xrightarrow{e} o$.

Besides input parameters that are consumed upon the invocation of a service operation, it might be the case that additional constraints must be met to successfully invoke an operation. These constraint are defined in the *assumption* of a service operation (see definition 5.8). Definition 5.16 formally introduces the term 'unsatisfied input'.

Definition 5.16 (Unsatisfied assumption) A constraint $c \in C \subseteq \Omega$ is an unsatisfied assumption of service operation $o \in OP$ if

$$c \in assump(o) \wedge c \notin world(s),$$

$$\text{with } s \in S \wedge e \in E \text{ such that } s \xrightarrow{e} o.$$

Particularly when constructing complex business processes, it can be difficult for the modeler to determine whether all steps in this business process are actually necessary for achieving the overall goal of the process. We say that an activity o is necessary for achieving the overall goal (or 'relevant'), if at least one output parameter produced by o is used by another activity p in the process. There must also be a path from o to p in the service composition, so that there is a data flow between both activities. The described properties do not need to be fulfilled if o is the last activity in the composition. Then, we assume that the activity is relevant. Definition 5.17 formally introduces the term 'relevance'.

Definition 5.17 (Relevance) A service operation $o \in OP$ in a service composition is relevant if

$$\begin{aligned}
&\big(\exists f \in post(o) \text{ such that} \\
&\exists p \in OP \text{ with } o \xrightarrow{e^*} p \\
&\wedge \ (f \in pre(p) \ \vee \ \exists g \in pre(p) \text{ such that } super^k(concept(f)) = concept(g)) \big) \\
&\vee \ \nexists q \in OP \text{ such that } o \xrightarrow{e^*} q.
\end{aligned}$$

Remark To denote that there is a path in a service composition connecting two vertices, we write

$$v_1 \xrightarrow{e^*} v_n, \ v_1, v_n \in V, e^* \subseteq E,$$

which is short for

$$v_1 \xrightarrow{e_1} v_2, v_2 \xrightarrow{e_2} v_3, \ldots, v_{n-1} \xrightarrow{e_{n-1}} v_n, \text{ with } v_1, \ldots v_n \in V, \ \{e_1, \ldots, e_{n-1}\} = e^* \subseteq E.$$

It might be the case that a service composition contains activities that produce the same output parameters. Such activities are potentially redundant. We speak of potential redundance, because there are circumstances that justify the fact that more than one activity produces an output parameter. However, it cannot be decided on the basis of semantic descriptions whether the inclusion of multiple activities producing equal postconditions is justified or not. Yet, it proves useful to inform the modeler in case potential redundancies are detected. The modeler can, then, investigate the issue and resolve the problem or, in case the activities are not redundant, flag the activities as explicitly not redundant. Definition 5.18 formally introduces potential redundance.

Definition 5.18 (Potential redundance) A service operation $o \in OP$ is potentially redundant if

$$\begin{aligned}
&\exists f \in post(o) \text{ such that} \\
&o \xrightarrow{e_1^*} s, \ e_1^* \subseteq E, \ s \in S \\
&\wedge \exists p \in OP \text{ with } p \xrightarrow{e_2^*} s, \ e_2^* \subseteq E \\
&\wedge \exists g \in post(p) \text{ such that } concept(f) = concept(g).
\end{aligned}$$

Remark While a service operation o is potentially redundant according to the above definition, it might, in fact, not be redundant: It is well possible that several facts, i.e. multiple instances of the same concept in the ontology, are required in a service composition. Whether or not this is necessary depends on the process that is being modeled. It cannot be decided by a semi-automated modeling envorinment at design time.

Based on the above definitions, we are, now, able to define the correctness criteria through which the semantic validity of a process can be decided. A definition of our understanding of the term 'semantic validity', as it will be used throughout the remainder of this thesis, is given in definition 5.19.

Definition 5.19 (Semantic validity) A service composition is semantically valid if

- it does not contain activities with unsatisfied inputs,
- it does not contain activities with unsatisfied assumptions,
- all activities in the composition are relevant, and
- it does not contain potentially redundant activities that have not been flagged by the user as explicitly not redundant.

The criteria for the semantic validity of a process should be checked after every action that the user undertakes in a semi-automated modeling environment. An agenda summarizing activities which do not satisfy all of the above criteria and what criteria they do not comply with should be kept. In doing so, the modeler has an overview over the problems with a composed service and can, subsequently, work on resolving them.

5.3.3 Possible Extensions

Keeping automatically track of the problems that exist regarding the activities in a service composition in the form of an agenda is a valuable support for the modeler. However, it is possible to provide additional support for the user by systematically searching for solutions to the problems on the agenda. In doing so, the semi-automated composition environment is able to suggest actions to the modeler that resolve the problems on the agenda.

In the following, we will describe possible actions for resolving problems that arise from the correctness criteria above in the case that they are not met.

- **Unsatisfied input:** If the service composition contains an activity that has an unsatisfied input, the semi-automated composition environment could suggest to:
  1. *Create a link:* The system can systematically traverse the service composition. Thereby, it can find out if there are any activities in the composition that produce the fact that is required by the activity with the unsatisfied input. The system can, then, suggest to create such a link that there is a data flow from the activity found to the activity with the unsatisfied input. In case there are multiple activities in the composition producing the required fact, the system can, accordingly, provide multiple suggestions, i.e. creating a link from each of the found activities to the activity with the unsatisfied input. However, these suggestions can be prioritized

and ordered: If there is an activity in the composition that is not relevant (according to definition 5.17), and if this activity produces the required fact, then, the suggestion to create a link from this activity to the activity with the unsatisfied input should be ranked first. Thus, two errors in the composition can be resolved with one single action.

2. *Insert a new activity:* No activity that is currently included in the service composition produces the fact that is required by the activity with the unsatisfied input. Then, the system uses semantic discovery to query the service repository for operations that produce the required fact as a post-condition. If such operations can be found in the service repository, the system suggests to include them before the activity with the unsatisfied input and to create a link between both activities. In case the semantic discovery process reveals more than one operation that produces the required fact, the system will, accordingly, make more than one suggestion. These suggestions can, then, be ordered in the following way: If the new operations that are suggested to be included in the composition will also yield unsatisfied inputs after the insertion, then, the suggested operations should be ranked on the basis of how many new problems of the type 'unsatisfied input' will be created by the insertion of the operations.

- **Unsatisfied assumption:** If the service composition contains an activity that has an unsatisfied assumption, the semi-automated composition environment could suggest to:

  1. *Create a link:* Analogously to the case where an activity has an unsatisfied input, the system can traverse the service composition and find activities that produce the required constraint as an effect.
  2. *Insert a new activity:* Analogously to the case where an activity has an unsatisfied input, the system can use semantic discovery to query the service repository for operations that produce the required constraint as an effect.

- **Irrelevant activity:** If the service composition contains an activity that is not relevant, the semi-automated composition environment could suggest to:

  1. *Create a link:* As described above when we discussed that creating a link can resolve the problem of an unsatisfied input of an activity, creating a link can also resolve the problem that an activity is not relevant: If there is an activity in the composition that is not relevant, and if this activity produces a fact required by another activity with unsatisfied inputs, then, the system suggests to create a link between both activities.
  2. *Remove the activity:* If there is no activity in the composition that requires facts produced by the irrelevant activity, then, the inclusion of this activity in the composition is not justified. The system, therefore, suggests to remove this activity from the composition.

- **Potential redundant activity:** If the service composition contains an activity that is potentially redundant, the semi-automated composition environment could suggest to:

    1. *Remove the activity:* If the modeler agrees that an an activity is redundant, it can be removed from the composition.
    2. *Flag activity as not redundant:* As discussed in definition 5.18 there might be the case that potential redundant activities are in fact not redundant. If this is the case, the modeler can manually flag the activity as not redundant and the problem is removed from the agenda.

5.4 Suggest Partial Plans

Users with various backgrounds and areas of expertise will apply different strategies when modeling a composed service. Particularly when modeling business processes, we can often identify parts of these processes which are more important than others, i.e. parts capturing the essence of a business process. These key parts in a service composition often comprise activities that reflect an area of expertise of the modeler. A person modeling a composed service might, on the other hand, not have a clear idea of what activities could be useful to wrap the key parts of the process to fit into a larger context. An example can be found in the following, where the Suggest Partial Plans mixed initiative feature will be motivated with our leave request example.

5.4.1 Business Scenario

The last step in our scenario was that the modeler resolved a problem with the *Check Create Leave Request* operation. When the user clicks on the operation to refresh the filtered list of available services, the tool will suggest the *Create Leave Request* operation. This is the last activity in the leave request process from the perspective of the user. However, the modeler might not be familiar with the fact that a specific check operation needs to be invoked in order to create a leave request in the system. It could have, therefore, been the case that the modeler directly selected the *Create Leave Request* operation from the list of all available services after the merge node depicted in figure 5.4. The modeler also creates the human activity producing the *TimePointPeriod* and links it to the *Create Leave Request* operation. Then, the modeler tries to create a link between the merge node of the parallel flow and *Create Leave Request*. The tool will detect that the set of postconditions up to the merge node does not satisfy the preconditions of *Create Leave Request* (the type *CheckCreateLeaveRequestResult* is missing). The tool instantly queries the semi-automated composition engine which detects that the insertion of the *Check Create Leave Request* operation would satisfy this open information requirement. The user is prompted whether or not the *Check Create Leave Request* should be inserted. The modeler approves this suggestion and the composition is complete.

5.4.2 Formal Description

Automated planners will always plan according to an algorithmic planning strategy, such as for example progression (i.e., forward search) or regression (i.e., backward search). Human planners will in contrast not always behave according to this schema when they model composed services. Users might have a clear idea about some specific activities that they want to include in the process, but possiblywithout a global understanding how the whole will fit together as a process.

A possible user behavior is to start modeling the part of the composed service that he or she is especially interested in, i.e. the modelers area of expertise. This is done by adding the respective operations and creating links denoting data flow between these operations. Then, they might continue with another key part of the composition, which does not directly follow the part of the composition the user has modeled first. This modeling strategy results in disconnected parts in a service composition that are likely to contain activities with unsatisfied preconditions or assumptions. In cases of this kind, it is desirable for the user to let the editor generate sub-processes that connect different parts of the composition, thereby satisfying the missing preconditions and assumptions.

Connecting two unrelated parts, i.e. a source part and a target part, in a composition constitutes a 'planning problem' in artificial intelligence. A planning problem consists of an initial state, a goal and a domain [22, 1]. Using our formal model for describing service compositions, the initial state $s_I \in S$ in a such planning problem would be a state as decribed in definitions 5.7 and 5.10, i.e. a set of facts and constraints. The domain is specified by our ontology Ω and the semantic descriptions of the operations in the service repository. The goal is a state $s_G \in S$ that consists of the preconditions and the assumptions of the first activities of the target part in the composition. The goal is formally described in definitions 5.20 and 5.21. Figure 5.6 gives an example of a service composition with two disconnected parts. It also contains two states labeled s_I and s_G, visualizing the initial state and the goal if the two parts were to be connected by a partial plan.

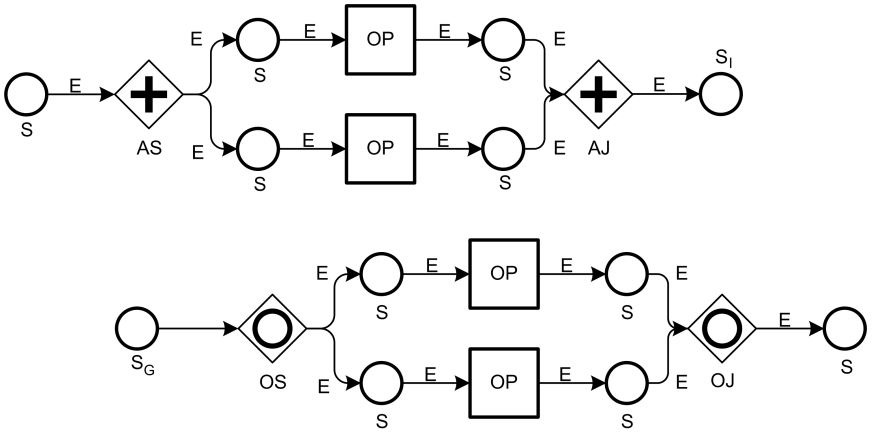

Figure 5.6: Two disconnected parts of a service composition

Definition 5.20 (Information state partition of the goal) The function $inf_G : S \rightarrow \mathcal{P}(F)$ returns the set of facts denoting the information state partition of a goal state s_G.

$$inf_G(s_G) = \begin{cases} pre(o) \\ \quad \text{if } \exists e \in E \text{ such that } s_G \xrightarrow{e} o,\ o \in OP \\ \\ \bigcup pre(o_i),\ \forall o_i \text{ such that } \exists d \in E \text{ with } v \xrightarrow{d} o_i \\ \quad \text{if } \exists e \in E \text{ such that } s_G \xrightarrow{e} v,\ v \in AS \\ \\ \bigcap pre(o_i),\ \forall o_i \text{ such that } \exists d \in E \text{ with } v \xrightarrow{d} o_i \\ \quad \text{if } \exists e \in E \text{ such that } s_G \xrightarrow{e} v,\ v \in OS \end{cases}$$

Remark For the sake of simplicity, we assume that the target part in the service composition does not start with a JOIN node of any kind or a node representing the beginning or end of a loop. The corresponding cases have, therefore, been ommitted in the definition.

Definition 5.21 (World state partition of the goal) The function $inf_G : S \rightarrow \mathcal{P}(C)$ returns the set of constraints denoting the world state partition of a goal state s_G.

$$world_G(s_G) = \begin{cases} assump(o) \\ \quad \text{if } \exists e \in E \text{ such that } s_G \xrightarrow{e} o,\ o \in OP \\ \\ \bigcup assump(o_i),\ \forall o_i \text{ such that } \exists d \in E \text{ with } v \xrightarrow{d} o_i \\ \quad \text{if } \exists e \in E \text{ such that } s_G \xrightarrow{e} v,\ v \in AS \\ \\ \bigcap assump(o_i),\ \forall o_i \text{ such that } \exists d \in E \text{ with } v \xrightarrow{d} o_i \\ \quad \text{if } \exists e \in E \text{ such that } s_G \xrightarrow{e} v,\ v \in OS \end{cases}$$

Remark For this definition the same assumptions are made as for definition 5.20.

The semi-automated modeling environment will suggest the insertion of a partial plan to connect the states s_I and s_G. Definition 5.22 formally introduces the notion of a partial plan.

Definition 5.22 (Partial plan) A partial plan connecting two states s_I and s_G in a service composition is defined as a service composition

$$C :=< V', E' >, \text{ with } V' \subseteq V,\ E' \subseteq E.$$

The initial state s_I is also the initial state s'_0 of the partial plan C ($s_I = s'_0$). There is a path connecting s'_0 and s_G, encompassing the actual partial plan:

$$s'_0 \xrightarrow{e^*} s_G,\ e^* \subseteq E'.$$

The partial plans generated by the system should, of course, be editable by the user of the semi-automated modeling environment.

5.4.3 Possible Extensions

When the semi-automated composition environment attempts to discover a partial plan that connects two states s_I and s_G, there will often exist multiple partial plans that have the desired properties, i.e. that produce the facts and constraints required in s_G. Then, the user has to select one of the alternative partial plans. A useful extension of the Suggest Partial Plans mixed initiative feature would be to introduce an ordering of these alternative partial plans when they are presented to the user for selection. Such ordering could be realized in two ways:

It is possibible to base the ordering of the alternatives upon the evaluation of the nonfunctional properties of the service operations in the partial plans. In doing so, the user could specify an attribute, e.g. cost, invocation time, around which the partial plans should be optimized. The optimization of a partial plan according to a nonfunctional property can be done by adding a corresponding requirement to the goal. The semi-automated composition engine can, then, use a planning strategy that is able to produce globally optimized plans using nonfunctional properties. This is done by providing aggregation functions for the different nonfunctional properties, e.g. the aggregation function for a nonfunctional property specifying the cost of the invocation of a service would be a basic *add* operation. This can also be done on the basis of multiple nonfunctional properties, in case the user has specified a weighting for the used combinations of nonfunctional properties. Various automated Web service composition approaches that take nonfunctional properties into account when producing a plan are available [29, 1, 33, 57, 43]. They could be used to realize this extension of the Suggest Partial Plans mixed initiative feature.

Another possibility to order the list of alternative partial plans is to use ratings of how often the user has selected the individual service operations in the alternative partial plans. These ratings can be aggregated and normalized for each alternative partial plan. This can be done by adding the rating values of all service operations in a partial plan. The sum is, then, divided by the number of service operations in the partial plan. This is done for each alternative partial plan. Similarly to the rating-based extension proposed for the Filter Inappropriate Service mixed initiative feature, the ordering of alternative partial plans does, in this case, not primarily build upon the use semantic service descriptions. It can, however, be realized with the use of a nonfunctional property in the semantic service descriptions that is used to track how often the modeler selects a service operation. The values can, then, serve as a basis for the rating.

6. Evaluation of Related Work in Semi-Automated Service Composition

In the following, an evaluation of the semi-automated composition approaches presented in section 3.2 will be given based on the supported mixed initiative features for semi-automated composition as well as additional criteria.

6.1 Evaluation According to the Mixed Initiative Features

Table 6.1 gives an overview of the mixed initiative features (as defined in chapter 5) that are supported by the semi-automated composition approaches presented in section 3.2, and to which extent they support them.

6.1.1 Support for Filter Inappropriate Services

Web Service Composer filters the list of services that can be included in the composition at each composition step. This realizes the feature 'Filter Inappropriate Services' that was presented in section 5.2. However, the realization of this feature in Web Service Composer is restricted in two ways:

First, the tool only considers OWL-S inputs and outputs, i.e. the mere data transformation that services realize. The assumptions that must be satisfied before the invocation of the services and the effects that the executions of the services have on the state of the world are not taken into account.

Second, the selection of appropriate services is done using a strict backward chaining approach: The user starts with the last activity in the process that he or she is creating. Then, the modeler has to select a service for every input of last service in the process so that all inputs are satisfied. This is, then, repeated for the newly added services until the composition is complete. This means in consequence that the plans constructed with the tool are not always optimal. For example, when one service operation delivers two outputs each of which satisfies a different input of

	Web Service Composer	**CAT**	**PASSAT**	**IRS-III**	**SSDC**
Filter Inappropriate Services	Does not consider assumptions and effects	N/A	N/A	Does not consider assumptions and effects	N/A
Extensions	Filtering on NFPs, ordering by degree of match	N/A	N/A	N/A	N/A
Suggest Partial Plans	N/A	N/A	HTN template expansion	N/A	Produces complete Guided Procedures, fully automatic approach, does not consider assumptions and effects
Extensions	N/A	N/A	High-level policies for composition	N/A	N/A
Check Validity	N/A	Evaluates 'well-formedness'	Tracks open information requirements	N/A	N/A
Extensions	N/A	Suggests fixes (ErrorScan)	Prioritized agenda	N/A	N/A

Table 6.1: Mixed initiative features supported by existing semi-automated composition environments

a downstream service, this services operation has to occur twice in the composed service.

Web Service Composer supports two extensions of the feature 'Filter Inappropriate Services' that were presented in section 5.2.3: First, the tool can further restrict the set of filtered services according to user-specified values of nonfunctional properties that are common to that set. Second, the list of filtered services which is presented to the user is ordered according to the 'goodness of match' as defined by Li and Horrocks [41]: Services that exactly produce a necessary input for a downstream service, i.e. an exact match, are ranked higher than services that produce outputs that subsume the necessary inputs.

IRS-III also supports the feature 'Filter Inappropriate Services'. While this feature in IRS-III has the same restrictions as Web Service Composer, none of the extensions specified in section 5.2.3 is realized.

CAT, PASSAT and SSDC do not support the 'Filter Inappropriate Services' mixed initiative feature.

6.1.2 Support for Suggest Partial Plans

PASSAT is the only tool of those included in this survey that partially supports the 'Suggest Partial Plans' mixed initiative feature. PASSAT is a tool for interactive plan authoring based on HTN networks. The user can invoke an automated planning mode to expand open tasks in the plan. This can be regarded as a specialization of the 'Suggest Partial Plans' feature in the sense that partial plans can only be generated from the current state to a state in which the composition is completed, i.e. all tasks can be executed. However, this realization of the feature is restricted in the way that the user must have completed the plan on a high level of modeling - otherwise the task network cannot be expanded.

In chapter 5 we have described a possible extension of the 'Suggest Partial Plans' feature: If there is more than one alternative for a partial plan, a ranking of user-specified nonfunctional properties should determine the order in which the alternatives are presented to the user. In PASSAT, the user can specify high-level policies, e.g. 'maintain an overall cost total of less than $100', which are also taken into account when automated template expansion is performed. This can be seen as a realization of that extension, as the alternative for a template expansion that conforms best to the specified policies will be presented to the user.

6.1.3 Support for Check Validity

PASSAT also supports the 'Check Validity' feature, as it interleaves a checking mechanism with the actual planning process: After each user action the system updates an agenda showing open information requirements that must be satisfied in order to have an executable plan. As an extension to this mechanism, PASSAT orders the agenda according to user-specified criteria.

Another, more thorough realization of the 'check validity' feature can be found in CAT. Here, the tool checks at each composition step if the composition complies with

a set of properties that describe the 'well-formedness' of the composition. In case these properties are violated, the system, consequently, presents a list of warnings and errors. As an extension of this feature, the authors present an algorithm that presents the user with suitable suggestions for further steps based on the evaluation of the well-formedness criteria.

Web Service Composer, IRS-III and SSDC do not support the 'check validity' mixed intiative feature.

6.2 Evaluation According to Additional Criteria

The mixed initiative features supported by a semi-automated composition approach are its most important characteristic. However, there are more criteria that allow further distinction among such approaches. In the remainder of this section, these criteria will be presented and applied to the semi-automated composition approaches presented in section 3.2. Table 6.2 shows an evaluation of the presented approaches according to these criteria.

An important criterion for the user who created composed services is the way in which composed services can be modeled with the system he or she utilizes. As human planners are likely to feel constrained when they are forced to adhere to an algorithmic planning strategy, the tools should give the users maximum freedom in modeling their compositions.

Web Service Composer and IRS-III impose a strict backward chaining planning strategy on the user. The user has to start with the last activity in the composition, i.e. the activity producing the desired end result. The inputs of this activity are, then, recursively satisfied until the first activity in the composition, e.g. a user input, is reached. Due to the strict backward chaining approach, only the last activities of compositions created with Web Service Composer and IRS-III can determine the end results, which is also problematic. In SSDC, the user has to specify the outputs of the overall process, which are, then, translated into a goal and automated planning is applied. The specification of a goal, however, is non-trivial and cannot be done intuitively by the user.

Another criterion that is highly important for the user of a semi-automated composition tool is whether or not a graphical user interface is provided. Web Service Composer, IRS-III and SSDC provide the user with a graphical user interface, while CAT and PASSAT merely offer textual modeling environments. Especially for complex compositions, the user can hardly oversee the causal relations among the activities.

Semi-automated service composition approaches rely on reasoning on domain knowledge that is specified in ontologies. In order to support the maintainability of the composed services that are created using semi-automated composition tools, standardized formats should be used for the ontologies. For the formal specification of domain knowledge presents a tremendous challenge, organizations have to rely on available ontologies that have been created by other parties as building blocks for assembling their domain knowledge. Web Service Composer and IRS-III build upon open formats such as OWL-S and WSMO. Furthermore, IRS-III allows the use of

	Web Service Composer	CAT	PASSAT	IRS-III	SSDC
Imposed planning strategy	Backward chaining	None	Top-down refinement	Backward chaining	Graphical goal specification
Modeling environment	Graphical	Textual	Textual	Graphical	Graphical
Knowledge base	OWL-S	Non-standard	Non-standard	WSMO	Based on OWL-S
Reasoning	Output-input subsumption	Output-input subsumption	Considers assumptions and effects	Output-input subsumption	Output-input subsumption
Control constructs	Not provided	Not provided	Not provided	Not provided, If-then-else construct	Sequence
Compositions are executable	Tool acts as Web service client	No	No	IRS-III orchestration engine	SAP NetWeaver
Output format	OWL-S process model	Non-standard	Non-standard	Non-standard	Non-standard

Table 6.2: Evaluation of existing approaches for semi-automated composition

WSMO mediators in the compositions, which eases the process of integrating ontologies from different parties. SSDC extends OWL-S in a way that Guided Procedures can easily be described. CAT and PASSAT, in contrast, are built upon proprietary formats for encoding domain knowledge.

When service capabilities and functionality are specified in ontologies, the data transformation effected by the invocation of a service can be specified as well as the change in the state of the world that the invocation of a service implies. Four out of the five semi-automated service composition tools presented in this survey only reason on the pre- and postconditions of the services that can be included in the compositions, i.e. the data transformation that the services effect. However, a large number of possible applications, i.e. the set of computational workflows, can be described using inputs and outputs only. PASSAT is the only approach among those presented here that explicitly supports constraints on the state of the world.

When modeling composed services, we would naturally expect the possibility to model control flow between the individual activities. Here again, four out of the five presented tools do not provide control constructs other than 'sequence'. This is probably related to the fact that most tools reason only on the inputs and outputs of the services that can be included in a composition: If preconditions and effects are not considered, the control flow of a composition can implicitly be derived from the data flow. PASSAT, being the only approach among those presented here supporting preconditions and effects, lacks a notion of explicit control flow. IRS-III, though, provides a basic if-then-else operator.

A semi-automated service composition tool should ensure that it is possible to execute the processes that can be modeled. This can be done either by directly providing the user with an execution environment or by exporting the compositions into an executable format. Web Service Composer allows to directly execute composed services by calling the individual services via the WSDL interface that is provided in the groundings of the OWL-S ontologies used. Since the tool acts as the Web service client for all calls, it does not support complex choreographies. IRS-III, on the other hand, comes with an orchestration engine on which the composed services can be enacted, allowing the user to specify choreographies that include more than two parties. SSDC produces processes that are encoded as SAP Guided Procedures, which can be deployed and executed on the SAP NetWeaver platform.

In addition to its Web service execution functionality, Web Service Composer is able to store the composed services as OWL-S process models. Together with an OWL-S grounding, the service compositions are also executable on platforms other than Web Service Composer. CAT, PASSAT and IRS-III do not offer the possibility to export composed services to any open format. The Guided Procedures produced by SSDC are also not an open format, but can be re-used in a SAP NetWeaver environment, at least. The de-facto standard format for executable service compositions is, however, WS-BPEL [19]. It is striking that none of the tools offers WS-BPEL export functionality.

7. Designing a Semi-Automated Composition System

In this chapter, we will describe the design of a semi-automated composer that is based on the mixed initiative features for semi-automated composition. We will start with developing the requirements for the composer in section 7.1. This motivates the development of a new system as opposed to extending one of the existing approaches for semi-automated composition. We will, then, describe how the mixed initiative features, that have been introduced in chapter 5, can be realized on top of a reasoner for semantic service descriptions.

7.1 Requirements Analysis

The purpose of this section is to outline the requirements for the semi-automated composer that has to be developed as a part of this work. Since we cannot guarantee to develop a complete set of requirements, we use the following approach to derive the requirements for the composer:

First, the requirements should be based on the mixed initiative features for semi-automated service composition that have been developed in chapter 5 of this thesis. The composer should, thus, support all three presented mixed initiative features. Chapter 5 also introduces possible extensions for each mixed initiative feature. The composer should be designed in such a way that these extensions can easily be added as optional requirements.

Second, the evaluation of existing semi-automated composition approaches in chapter 6 are to serve as a source of requirements for the semi-automated composer. Table 6.2 uses a set of distinctive criteria to compare the existing approaches. These criteria are used to formulate the following requirements:

- **No stipulation of a specific modeling strategy.** When modeling a composed service, the user should not be forced to adhere to any predefined strategy. Some of the approaches presented for semi-automated composition force

the modeler to adhere to a strategy that is implied by the underlying realization. This results in a situation where the modeler has to follow a backward-chaining or strict top-down refinement approach, or has even to specify the goal of the process in terms of information artifacts that are to be produced. The composer that is to be developed should be an intuitive modeling environment and, thus, not impose any specific modeling strategy.

- **Provision of a graphical user interface.** The creation of a service composition is a difficult task, because the modeler has to keep track of all the dependencies existing between the individual operations in the composition. The modeling must, therefore, be supported by a graphical user interface.
- **Use of open standards for semantic markup.** To ease the development of the semantic descriptions of the services to be composed and to exploit publically available ontologies and reasoning systems to the largest possible extent, the semantic markup is to be based on an open standard. Considering the results of the evaluation of the frameworks for semantic service descriptions presented in section 3.1.6, either WSMO [54] or OWL-S [42] should be used. This is due to the maturity and prevalence of these approaches, as well as the availability of reasoners for both languages.
- **Reasoning on complete service capabilities.** To exploit the full potential of semantic service descriptions, the composer should take assumptions and effects into account when reasoning on the descriptions.
- **Provision of adequate control constructs.** The existing approaches for semi-automated composition do not provide the user with an adequate set of control constructs. This is possibly due to the fact that these approaches have evolved from automated planning, where a large variety of control constructs is not important. The modeling of business processes, though, does not make sense without having control constructs available. Ideally, the composer that is to be developed would support all the workflow patterns that have been described by van der Aalst et al. in [67]. This would, however, be beyond the scope of this work. The requirement for the composer is, therefore, to support parallel and conditional flow as well as basic loops. This is also in line with the formal model for service compositions presented in section 5.1, upon which the mixed initiative features have been formally defined.
- **Ability to export to a common process language.** The composed services should be re-usable in other environments. It should, thus, be possible to export composed services into a common format for process descriptions. This is useful when the composed services is to be executed by a BPMS, for example. Here it would be optimal to use WS-BPEL [19], which is the de-facto standard for executable process descriptions.

Based on the above requirements and the requirement that the semi-automated service composer to be developed should support all three mixed initiative features, it has to be discussed whether one of the existing approaches should be extended or

if a new system should be developed. The evaluation of related work in chapter 6 has revealed that none of the presented semi-automated modeling tools is complete in terms of functionality. None of the presented existing approaches covers all the mixed initiative features that have been developed in chapter 5. The presented approaches could hardly be extended in order to provide the missing functionality. Furthermore, the requirements that have been derived from the evaluation in chapter 6 would make it very difficult to extend one approach in order to satify all of the requirements. What's more, the presented approaches are not publically licensed and no sources are available. As the work presented in this thesis is done at SAP, it would be possible to build upon the Smart Service Discovery and Composition (SSDC) [53] project. However, among the approaches that were presented, SSDC is the least complete in terms of supported mixed initiative features. Therefore, a new semi-automated service composer will be developed.

7.2 Architectural Considerations

A new semi-automated service composer has to be designed based on the above requirements. The mixed initiative features are quite similar to known applications in the field of semantic service discovery [53, 61, 41, 32]. Such discovery mechanism are, usually, built on top of a reasoner. It would, therefore, make sense to implement the mixed initiative features as an extension of an existing reasoning environment. The reasoner that has been developed in the ASG project [37] is publically licensed under the GNU Lesser General Public License[1]. Furthermore, the reasoner works with WSML [16] service descriptions and WSMO [54] ontologies, which are open standards for semantic markups. It is, therefore, chosen as the reasoner that will be extended in order to support the mixed initiative features. In its current implementation, however, the reasoner does not support assumptions and effects. Also, disjunction in logical expressions is not supported. Yet, the interfaces of the reasoner include assumptions and effects and the reasoner is still under development. In future versions of the reasoner, these features are likely to be supported. Another reason underpinning the selection of the ASG reasoner is that a composition component is available for this reasoner. As we will see in section 7.5, the 'Suggest Partial Plans' mixed initiative feature can be realized on top of the composer.

Also based on the requirements that were defined above, it is necessary to provide a graphical modeling environment that uses the functionality provided by the extended reasoning component. Based on the evaluation of the various modeling tools used within SAP, which was presented in section 2.2.1, we have chosen Visual Composer as the front-end for our mixed initiative functionality. This is because Visual Composer in currently used as the prototyping environment for the Galaxy project, where SAP bundles its efforts towards creating a fully-fledged business process engine. One of these prototypes extends Visual Composer with the capability to model business processes using BPMN. The fact that a graphical modeling language with a rich set of control constructs was one of the criteria for semi-automated modeling

[1] See `http://www.gnu.org/licenses/lgpl.html`

environments discussed in section 6.2 further underpins the decision for Visual Composer. Throughout the remainder of this chapter, we will focus on the realization of the semi-automated composition component that is realized on top of the reasoner.

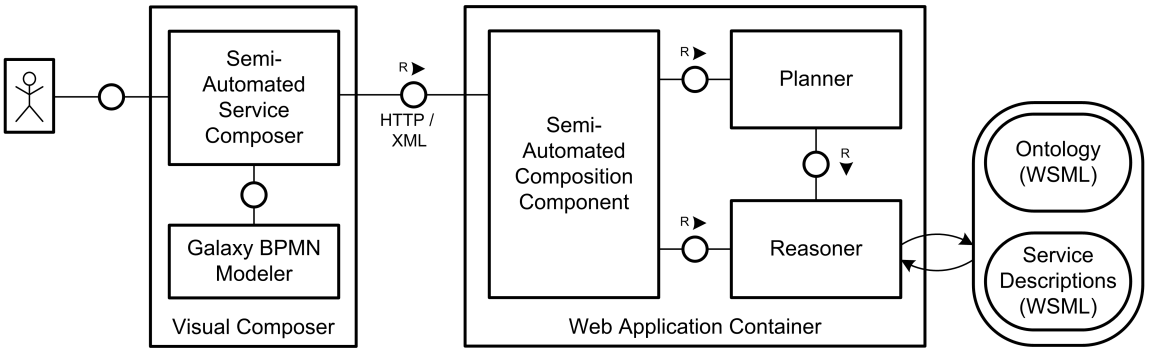

Figure 7.1: The environment of the semi-automated service composer

Figure 7.1 shows the environment of the semi-automated service composer using the Fundamental Modeling Concepts (FMC) block diagram notation [35]. The user accesses the front-end of the semi-automated service composer, which is realized in Visual Composer. The service composer sends queries to the backend, with which it exchanges serialized data in XML via HTTP. The backend consists of three main components: The semi-automated composition component, the reasoner and the planner. The functionality in terms of the mixed initiative features is provided by the semi-automated composition component. The component contains the algorithms that are used to realize the mixed initiative features. These algorithms are designed in a way that they use the reasoner and the planner. A typical query of the semi-automated composition component to the reasoner would, for example, be 'does a subsume b?'. Only the mixed initiative feature 'Suggest Partial Plans' uses the planner. The other two mixed initiative features rely solely on the reasoner. The dynamics among the components of the semi-automated service composer are also depicted in figure 7.2 using the UML 2.0 Sequence Diagram notation [50].

Throughout the remainder of this chapter, it will be discussed how each of the three mixed initiative features is realized. The mixed initiative features Filter Inappropriate Services and Suggest Partial Plans are realized solely in the semi-automated composition component in the backend. The mixed initiative feature Check Validity is realized partly in the Visual Composer front-end.

7.3 Realization of Filter Inappropriate Services

The semi-automated service composition component contains two methods that realize the mixed initiative feature 'Filter Inappropriate Services', as shown in listing 7.1. The method `findInvocableServicesOrdered` returns the list of service operations that are *invocable* in the supplied state, according to definition 5.11. The method `findNearlyInvocableServices` returns a list of service operations that are *nearly invocable* in the supplied state, according to definition 5.12. The argument *state* that is required by both methods is a set of facts $inf(s)$ that are available in state s (see also definition 5.7).

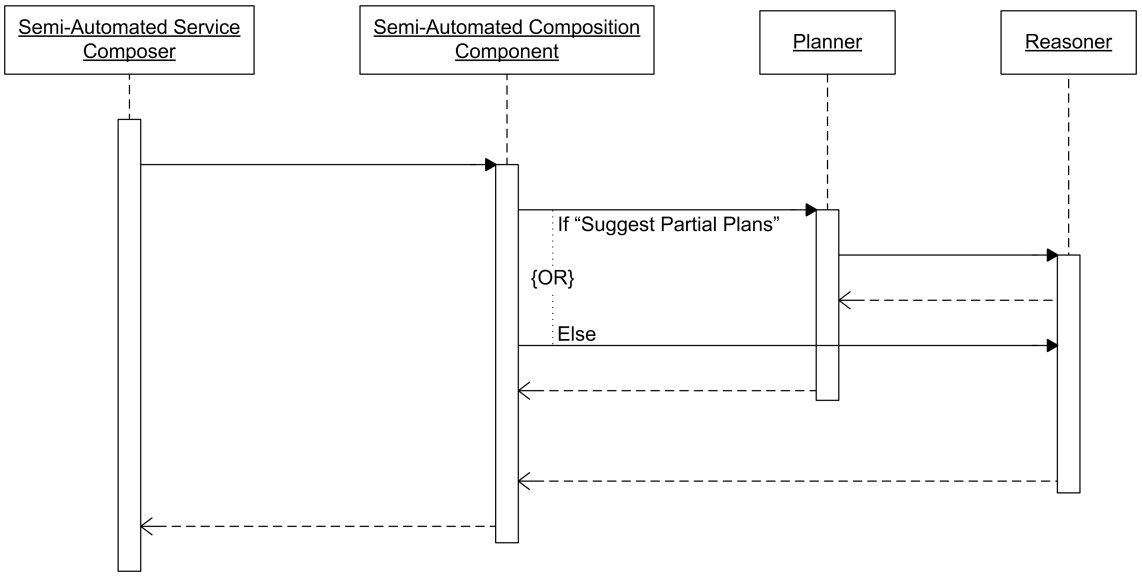

Figure 7.2: Interactions among the different components of the semi-automated composer

```
1 /**
   * Returns a list of all service operations that are invocable in the supplied state
3  */
  public Vector<ExecutableServiceOperation>
5           findInvocableServicesOrdered( State state, String role )

7 /**
   * Returns a list of all service operations that are nearly invocable
9  * in the supplied state
   */
11 public Vector<NearlyExecutableServiceOperation>
           findNearlyInvocableServices( State state )
```

Listing 7.1: Java interface of ‘Filter Inappropriate Services’

Both invocable and nearly invocable service operations are represented by corresponding classes. Figure 7.3 depicts the different kinds of service operations known to the semi-automated composition component using UML class diagram notation [50]. The abstract class `ServiceOperation` contains the basic attributes of the service operations as well as getter and setter methods for these attributes. They are the name of the operation, the service containing the operation, i.e. a port type on the WSDL level or an enterprise service in the context of SAP, its pre- and postconditions, and the roles that the operation supports. These roles reflect a nonfunctional property in the semantic descriptions of the operations specifying the roles for which an operation has been designed. An operation might be designed for several roles; therefore, there is also an attribute specifying the intended or default role of the operation. A service operation can either be invocable, nearly invocable or not invocable in a given state. The three classes `InvocableServiceOperation`, `NearlyInvocableServiceOperation`, and `NonInvocableServiceOperation` extend the

abstract class `ServiceOperation`. They contain additional attributes (along with the respective getter and setter methods) that are specific to whether a service is (nearly) invocable or not. In our current implementation, however, only the class `InvocableServiceOperation` requires additional attributes.

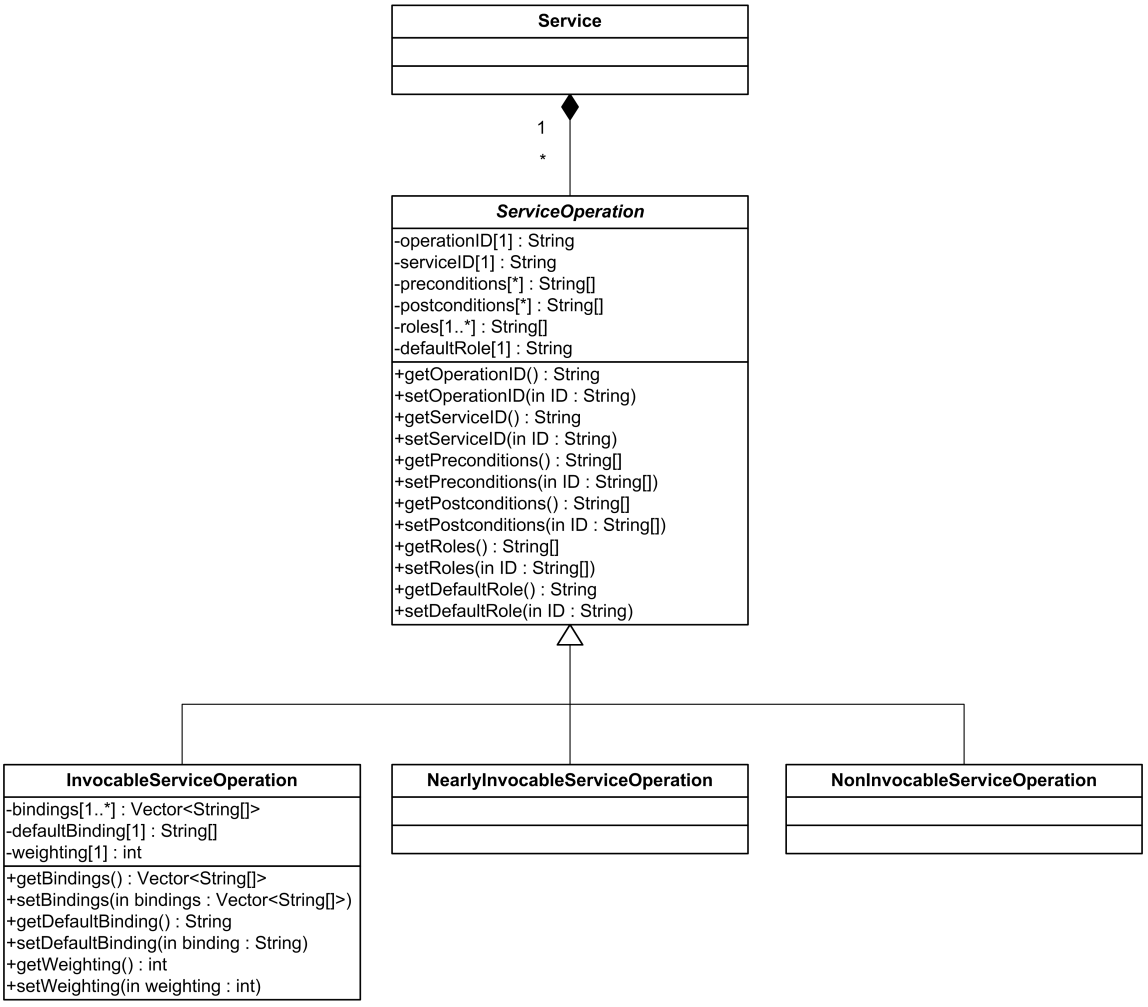

Figure 7.3: Different kinds of service operations

7.3.1 The method `findInvocableServicesOrdered`

The first method returns a list of invocable service operations, i.e. a list of objects of the class `InvocableService`. As described above, an invocable service operation has several attributes: The pre- and postconditions, possible roles in which they can be executed and a default role. In addition, an invocable service operation has one ore more bindings. The binding of an invocable operation consists of facts (see definition 5.3) for all of its input parameters, i.e. the precondition. However, it might happen that multiple available facts can satisfy a particular input parameter. Therefore, an invocable operation has as many bindings as there are different combinations of facts corresponding to the individual input types. All these bindings are stored with the the invocable operation. Listing 7.2 presents our algorithm for finding and ranking

invocable services for a given state in pseudo code. The algorithm will be discussed in the following.

```
findInvocableServicesOrdered(State state, String role) {
  register state with reasoner;
  retrieve list of registered operations from reasoner;
  for each registered operation do {
    if (operation is invocable) do {
      compute total match distance for each binding;
      store binding with lowest total match distance as default binding;
      store all other bindings, preconditions and postconditions;
    }
  }
  unregister state with reasoner;
  compute weightings based on match distances;
  for each invocable operation do {
    if (NFP specifying an intended roles for the operation exists) do
      if (NFP matches role) do
        increase weighting by 1;
  }
  order operations by weighting;
  return ordered list of operations;
}
```

Listing 7.2: Compute ordered list of invocable service operations

First, the information state that has been supplied by the frontend is passed to the underlying reasoner. Then, a list of all service operations which are currently registered at the reasoner is retrieved. This list is, then, iterated. The underlying reasoner is queried for each registered operation to decide whether it is invocable in the registered state or not. If the operation is invocable, the reasoner returns all bindings in which the operation is invocable. Afterwards, the 'total match distance' is calculated for every binding of the operation (line 6). The total match distance of a binding is the sum of the match distances (see definition 5.12) between the facts in the supplied state and the precondition. The match distance is computed based on the distance between a concept represented by a fact in the binding and the corresponding fact in the preconditions of the operation.

An example is depicted in figure 7.4. We reconsider the operation *Read Employee Time Account* that has been discussed in section 4.1. As a precondition, this operation consumes one parameter of the type `Date` and another parameter of the type `Employee`. Our ontology consists of a concept `Date`, as well as of three concepts `Person`, `Employee` and `Manager`, where `Employee` is a subconcept of `Person`, and `Manager` is a subconcept of `Employee`. We assume a current state that contains three facts: A fact 'dat' corresponding to the concept `Date`, a fact 'emp' corresponding to the concept `Employee`, and a fact 'mgr' corresponding to the concept `Manager`. In this state, the operation is invocable with two different bindings. The first binding contains the facts 'dat' and 'emp'. The match distance between the fact 'emp' and the concept `Employee` is zero. The second binding contains the facts 'dat' and 'mgr'.

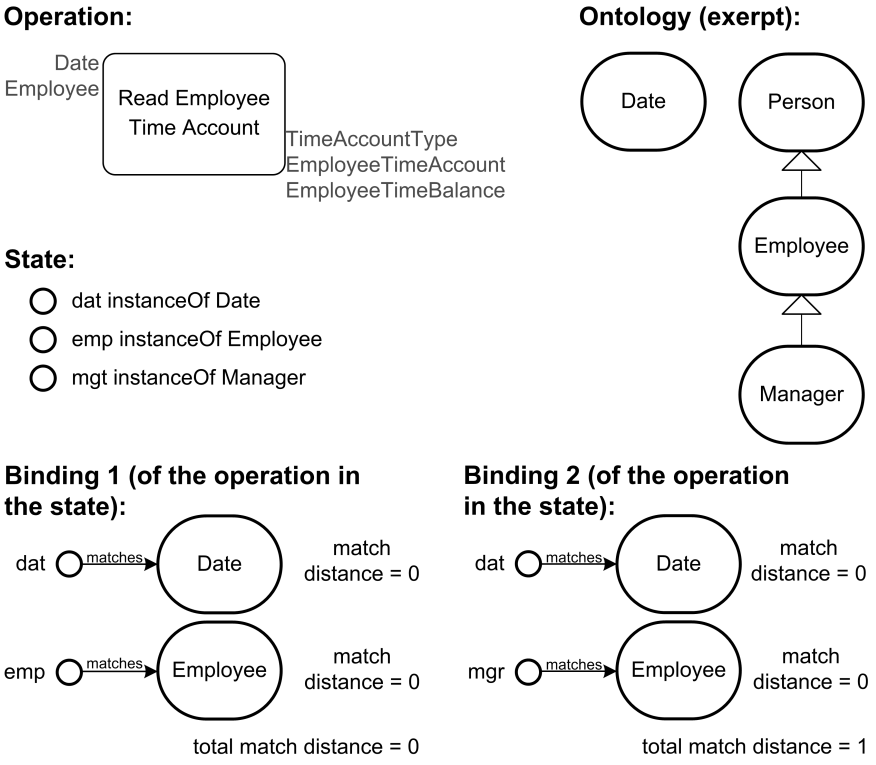

Figure 7.4: Example illustrating the concepts 'match distance' and 'total match distance'

The match distance between the fact 'mgr' and the concept `Employee` is one: While 'mgt' is not a direct instance of the concept `Employee`, it can be derived from it, because `Manager` is is a subconcept of `Employee`. In both bindings exists a fact 'dat' corresponding to the type `Date`. This fact has a match distance of zero in both cases. Hence, the total match distance, i.e. the sum of all match distances in a binding, of the first binding is zero. The match distance of the second binding is one.

After the total match distance has been computed for all bindings of the current operation in line 6 of listing 7.2, the default binding is selected. The default binding is the binding of an operation that has the lowest total match distance. In the above example, the first binding of the operation has a lower match distance than the second one. Therefore, it is the default binding of the operation in this example. The default binding is stored together with the other bindings and the pre- and postconditions of the operation. After the iteration of the loop for each invocable service, a list of invocable services in the supplied state has been built and the state is unregistered with the reasoner. This list is now to be ordered according to the relevance of the individual invocable operations for the user. This step is carried out in line 12 of listing 7.2, and is realized using the following weighting algorithm:

Each invocable operation is assigned a weighting based on the total match distances of its bindings: We, first, select the binding with the lowest match distance, i.e. the default binding, for each operation. In doing so, we obtain the value of the lowest total match distance for each operation. On the basis of this value, a weighting is assigned to each operation. The operation with the highest value is assigned the

lowest weighting, i.e. zero; the operation with the second highest value is assigned the weighting one, and so on. Our experiments have shown that situations in which multiple operations have the same minimum match distance occur frequently. To further differentiate the weighting, roles are taken into account: In the frontend of the semi-automated composition environment, where the invocation of the method `findInvocableServicesOrdered` is triggered, swimlanes are used to visualize the role in which the operations residing within the lane are to be invoked at runtime. This role is passed to the semi-automated composition component together with the information state at the selected point in the composition when `findInvocable-ServicesOrdered` is called. In the weighting algorithm, this role is compared to the intended role of each invocable operation. An intended role is the role for which an operation has been designed. For example, the operation *Create Leave Request* is designed for the role 'employee'. However, it is also invocable in the role 'manager', because the roles correspond to concepts in the ontology and `Manager` is a subconcept of `Employee`. The intended role is specified as a nonfunctional property that is part of the semantic descriptions of the service operations. If the role supplied by the frontend matches the intended role of an operation, the weighting of this operation is increased.

7.3.2 The method `findNearlyInvocableServices`

The second method provided by our semi-automated composition component discovers service operations that are nearly invocable to the degree $k = 1$ in a given state, according to definition 5.12. An operation is nearly invocable to the degree $k = 1$, if only one input parameter is not satisfied in the supplied stae. Listing 7.3 shows the pseudo code of this method. In order to find operations missing just one input type in order to be invocable, we traverse the ontology, add concept after concept to the current state, and query the underlying reasoner for invocable operations in that modified state. We assume that all nearly invocable service operations are equally relevant. Consequently, there is no ranking among nearly invocable services. This allows for an optimization of the traversion of the ontology: Not every concept has to be added to the supplied state so that the reasoner can be queried to return invocable operations. It is sufficient to perform these queries only for the most specific subconcepts of each traversed concept in the ontology. The notion of the most specific subconcept is formally introduced in definition 7.1.

Definition 7.1 (Most specific subconcept) A concept can be seen as a node in the hierarchichal structure of the concepts defined in the ontology. The most specific subconcepts of a concept are the leave nodes in this hierarchy that are reachable from the concept by traversing its subconcepts. If a concept does not have any subconcepts, then, it is already a most specific subconcept. A concept that has subconcepts can have one or more most specific subconcepts. The function mss formally describes the notion of the most specific subconcept.

$$mss : C \rightarrow C$$
$$mss(c) := d \in sub(c) \text{ such that } sub(d) = \emptyset$$

Example Reconsidering the example from figure 7.4, `Manager` is the most specific subconcept of `Person`.

The most specific subconcepts of a concept can be found using a recursive algorithm, which is also shown in listing 7.3. After a most specific subconcept is added to the state, the underlying reasoner will also discover the operations that would be invocable if the superconcepts of the most specific subconcept were part of the state. Using this optimization, the number of search operations performed against the reasoner can be reduced. In our prototypical tool, the response time could be improved significantly after the implementation of this optimization.

```
findNearlyInvocableServices(State state) {
  operations = findInvocableServices(state);
  for each concept in the ontology do {
    if (concept is not marked as visited) do {
      sc = findMostSpecificConcepts(c);
      for each concept s in sc {
        add s to state;
        register state with reasoner;
        nOps = findInvocableServices(state);
        add nOps − operations to result;
        deregister state with reasoner;
        remove s from state;
        mark s as visited;
      }
    }
  }
  return list of nearly invocable operations (result);
}

Concept[] findMostSpecificConcepts(Concept c) {
  if (c has subconcepts) do {
    sc = new Concept[];
    for each subconcept s of c
      sc += findMostSpecificConcept(s);
    return sc;
  }
  else return [c];
}
```

Listing 7.3: Compute list of nearly invocable service operations

7.4 Realization of Check Validity

For the realization of the mixed initiative feature Check Validity as defined in section 5.3.2, the backend component should, ideally, provide methods to find unsatisfied inputs and assumptions of specific service operations in a composition as well as irrelevant and redundant service operations. However, we chose to realize Check

Validity directly in the front-end, i.e. the modeling environment. This is because the underlying WSML reasoner is - in its current version - unable to evaluate assumptions and effects in semantic service descriptions. Therefore, only the pre- and postconditions of the operations are taken into account for the realization of Check Validity. The information about the required input parameters for the operations in a composition, however, is known to the front-end. The operations currently in the composition and the information on how they are linked to each other are also known to the front-end. That means that all the necessary information to realize Check Validity is available in the front-end. Furthermore, Check Validity is an interactive feature that is supposed to visualize problems with the composition just at the point in time when they arise, i.e. when the modeler changes the composition. Therefore, round-tripping with the backend is not desirable when it is not absolutely necessary. By realizing this mixed initiative feature in the front-end, it can be ensured that the system responds instantly to changes by the modeler. In the following, the realization of Check Validity is presented in detail.

7.4.1 Detecting Unsatisfied Inputs

Information about operations that have unsatisfied inputs can be obtained by traversing the graph representing the composition. Listing 7.4 lists the pseudo code for detecting unsatisfied inputs of the operations in the composition. The algorithm will be discussed in the following.

```
findUnsatisfiedInputs(CompositionGraph comp, Role role) {
  for each operation op in comp do {
    requiredInputs = preconditions of current operation op;
    availableTypes = {role};
    links = links connected to incoming plugs of op;
    for each link in links do
      recurseLink(link);
    unsatisfiedInputs = requiredInputs − availableTypes;
    if (unsatisfiedInputs != ∅)
      store unsatisfiedInputs with current operation;
  }
}
```

Listing 7.4: Detecting unsatisfied inputs in the service composition

The method `findUnsatisfiedInputs` builds a list of available types for every operation in the composition. Referring to the model of a service composition as introduced in definition 5.1, the list of available types for an operation consists of all facts that are available in the state directly preceding the operation. The organizational role, in which an operation is invoked, is also in the list of available types of an operation. As opposed to the formal, conceptual model of a service composition, the implementation of the composition graph in the front-end does not explicitly contain states. Instead, the lists of available types are computed by traversing the composition graph. Possible node types in the composition graph are operations, AND-splits and -joins, OR-splits and -joins, as well as nodes representing the beginning or the end of a loop. The composition graph also contains a start node.

The nodes can be connected using links. Every node can have multiple incoming and outgoing links, except for the start node, which cannot have incoming links. `findUnsatisfiedInputs` carries out the following steps for each operation in the composition: First, it is determined what inputs are required so that the operation is invocable. Therefore, all the facts part of the precondition of the operation are added to the list of required types. Second, the list of facts that are available at the point of the composition exactly before operation is determined. Therefore, the role of the operation is added to the list of available types. Then, the incoming links of the operation are traversed, thereby adding the postconditions of all preceding operations to the state. This is done using a recursive algorithm which is implemented in the method `recurseLink` (see listing 7.5). After the list of available types has been built, it can be determined whether or not the operation has unsatisfied inputs by examining the difference between the required inputs and the available types. This is done in line 8 of listing 7.4. If there are unsatisfied inputs, they are, then, stored and an event is generated that triggers the visualization of the problems.

```
recurseLink(Link link) {
2   currentNode = link.source;
    if (currentNode == start node) do
4     return;
    if (currentNode is a service operation) do
6     availableTypes += currentNode.postconditions;
    links = incoming links of currentNode;
8   if (currentNode is an OR−join) do {
      availableTypesOR[] = new Array;
10    i = 0;
      for each link in links do {
12      availableTypesOR[i] = ∅;
        recurseOR(link, availableTypesOR[i]);
14      i++;
      }
16    availableTypes += {availableTypesOR[0] ∪ ··· ∪ availableTypesOR[i]};
    } else {
18    for each link in links do
        recurseLink(link);
20  }
}
```

Listing 7.5: Recursive traversion of the composition graph

Listing 7.5 provides more detail on how the composition graph is traversed in order to obtain the set of available facts. The method `recurseLink` takes a link connecting to nodes in the composition graph as an input. The method is called by `findUnsatisfiedInputs`, which always supplies an incoming link of the current operation as input (see line 5 of listing 7.4). Therefore, selecting the node connected to the source of the supplied link (see line 2 of listing 7.5) always means selecting the node in the composition graph that directly precedes the current node. In doing so, the graph can be traversed from the current node backwards to the start node. During this traversion, the list of available types for the current operation in the

calling `findUnsatisfiedInputs` method is built. This is done by adding the facts in the postconditions of the traversed operation nodes to the list. In general, the set of available types for an operation is the conjunction of the postconditions of the preceding operations on the path between the operation and the start node. When traversing OR-constructs, however, only the intersection of the postconditions of the operations on the different paths between the OR-split and the OR-join is added to the list. This satisfies a requirement from definition 5.7: Ideally, only the postconditions of the operations on the OR-path (or the OR-paths, in the case of an inclusive OR-construct) that is selected at runtime should be added to the set of available types. As our semi-automated composition approach is used at design-time, only those facts are added to the set of available types that appear in at least one postcondition on all paths of the OR-construct. This intersection of postcondition is built using a separate recursive method, which is shown in listing 7.6.

```
1  recurseOR(Link link, Array availableTypesOR[i]) {
     currentNode = link.source;
3    if (currentNode is an OR−split) do
       return;
5    links = incoming links of currentNode;
     if (currentNode is an OR−join) do {
7      availableTypesNestedOR[] = new Array;
       j = 0;
9      for each link in links do {
         availableTypesNestedOR[j] = ∅;
11       recurseOR(link, availableTypesNestedOR[j]);
         j++;
13     }
       availableTypes +=
15       {availableTypesNestedOR[0] ∪ · · · ∪ availableTypesNestedOR[j]};
     }
17   if (currentNode is a service operation) do
       availableTypesOR[i] += currentNode.postconditions;
19   for each link in links do
       recurseOR(link);
21 }
```

Listing 7.6: Recursive traversion of the composition graph in the special case of an OR-join

The method `recurseOR` is similar to `recurseLink`, except that `recurseOR` needs to work with call-by-reference in order to deal with nested OR-constructs. A set of available types is built for each path of an OR-construct. The intersection of the available types is, then, first done for all paths of a nested OR-construct, which might itself be on an alternative path of a parent OR-construct.

7.4.2 Detecting Irrelevant Operations

In order to detect operations in a service composition which are irrelevant according to definition 5.17 the algorithm, shown in listing 7.7, has been constructed. Similar to the algorithm detecting unsatisfied inputs, which has been described above, the composition graph is recursively traversed for this detection.

```
1  findIrrelevantOperations(CompositionGraph comp) {
     for each operation op in comp do {
3      isRelevant = true;
       outLinks = links connected to outgoing plugs of op;
5      for each link in outLinks do
         if ((link.target ≠⊥) ∨ (link.target ≠ end node)) do
7          isRelevant = false;
       if (!isRelevant) do {
9        for each link in outLinks do {
           isRelevant = recurseFw(post(op), link);
11         if (isRelevant) do
             break;
13       }
       }
15     if (!isRelevant) then mark op accordingly
     }
17 }

19 boolean recurseFw(Fact[] pc, Link link) {
     currNode = link.target;
21   if ((currNode =⊥) ∨ (currNode = end node)) do
       return false;
23   outLinks = links connected to outgoing plugs of currNode;
     if ( (currNode is an operation)
25     ∧ (an element of pc is also in post(currNode)) ) do
       return true;
27   for each link in outLinks do {
       isRelevant = recurseFw(pc, link);
29     if (isRelevant) do
         break;
31   }
     return isRelevant;
33 }
```

Listing 7.7: Detecting irrelevant operations in the service composition

Definition 5.17 requires that, at least, one fact in the postcondition of a service operation must be present in a precondition of any other service that is reachable from that operation. As this needs to be checked for each operation in the composition, the algorithm begins with an iteration of all the operations. The only operation that is allowed to violate the relevance property is the last operation in the composition, which is always considered relevant. This exception is realized in line 6 of listing 7.7. If the current operation is not the last operation in the composition, then all the outgoing links of the operation are traversed using a recursive function (line 10).

The method `recurseFw` (also shown in listing 7.7) takes two parameters as inputs. The first parameter is a list of facts, constituted by the postcondition of the operation which is currently checked for relevance. The second parameter is an outgoing link of the node in the graph that is currently under investigation. The first step of

`recurseFw` is, therefore, to check the target node of the supplied link. If the node does not exist or if it is the end node of the composition, then, `recurseFw` will return *false* (line 22). However, the operation under investigation might still be relevant, as `recurseFw` is called for every outgoing link of the operation. If the target node of the supplied link is an operation, the algorithm checks if a fact in this operation's precondition is also contained in the list of facts supplied to `recurseFw`, i.e. the postcondition of the operation under investigation. If this is the case, the method will return *true* (line 26), stating that the operation under investigation is relevant. In all other cases, the outgoing links of the current node are recursively traversed (line 28), until either an operation with a precondition containing a fact from the supplied list is found or the end of the composition is reached.

7.4.3 Detecting Potentially Redundant Operations

The algorithm shown in listing 7.8 detects potentially redundant operations in a composition, according to definition 5.18.

```
1  findPotentiallyRedundantOperations(CompositionGraph comp) {
     for each operation op in comp do {
3      isPotentiallyRedudant = false;
       outLinks = links connected to outgoing plugs of op;
5      for each outLink in op.outLinks do {
         nextNode = outLink.target;
7        while ((nextNode ≠⊥) ∨ (nextNode ≠ end node)) do {
             for each inLink in op.inLinks do {
9              if (inLink ≠ outLink) {
                 isPotentiallyRedudant = recursePr(post(op), inLink);
11               if (isPotentiallyRedudant) do
                   break;
13             }
             }
15           if (isPotentiallyRedudant) do
               break;
17       }
       }
19     if (isPotentiallyredundant) then mark op accordingly
     }
21 }
```

Listing 7.8: Detecting potentially redundant operations in the service composition

Similar to the algorithms discussed above, the algorithm for the detection of potentially redundant operations begins with an iteration of all operations. For each operation, `findPotentiallyRedundantOperations` will traverse all nodes in the composition graph which are on a path between the operation and the end node or the last operation in the composition (line 7 of listing 7.8). For all traversed nodes it is checked whether the node has other incoming links than the one through which it was reached by the algorithm. If this is the case, a recursive function is called for each link. This function determines whether there are any operations on the

path between the link to the start node that produce facts which are also contained in the postcondition of the operation currently under investigation. The recursive function is shown in listing 7.9. It is similar to the recursive methods which have already been discussed, and will, therefore, not be explained here in further detail. A problem with realizing `findPotentiallyRedundantOperations` in the presented way is that the algorithm is likely to produce a high recursion depth. For future versions of our semi-automated composition engine, we will, therefore, investigate the possibilities of using optimized, less computing-intensive techniques for traversing the composition graph.

```
1  boolean recursePr(Fact[] pc, Link link) {
     currentNode = link.source;
3    if (currentNode == start node) do
       return false;
5    if ( (currNode is an operation)
       ∧ (an element of pc is also in post(currNode)) ) do
7      return true;
     links = incoming links of currentNode;
9    for each link in outLinks do {
       isPotentiallyRedundant = recursePr(pc, link);
11     if (isPotentiallyRedundant) do
         break;
13   }
     return isPotentiallyRedundant;
15 }
```

Listing 7.9: Recursive traversion of the composition graph for detecting potentially redundant operations

7.5 Realization of Suggest Partial Plans

As opposed to the realization of the Check Validity mixed initiative feature, Suggest Partial Plans is realized in the backend component. Suggest Partial Plans is the only mixed initiative feature that uses the composer component depicted in figure 7.1 (see page 72). The composer has been developed as a part of the ASG project. The planning algorithm that we use with the composer is the result of Harald Meyer's diploma thesis [43]. In the following, it will be discussed how the semi-automated composition component builds on top of this composer to realize Suggest Partial Plans.

In the formal introduction of this mixed initiative feature in section 5.4.2, we have identified the initial state, a goal and a domain as the constituents of a planning problem. The problem of suggesting partial plans to a user, i.e. connecting to unrelated operations in the composition, can be translated into a planning problem: The initial state contains all the available types of the source operation and the facts in its postcondition. When discussing the realization of Check Validity, it has already been described how the available types of an operation can be computed. The algorithm is shown in listing 7.4. By adding the postconditions of the source

operation for Suggest Partial Plans to its available types, the initial state of the planning problem is complete. The goal of the planning problem corresponds to the precondition of the target operation. The domain consists of the specified service operations and the ontology, which are registered with both the reasoner and the composer.

Given a concrete planning problem, the task of the composer is to produce a plan leading from the initial state to the goal. In order to do so, the composer uses a planning algorithm. A discussion of planning algorithms that could be used in this context would be beyond the scope of this work. An overview of the most impotant planning algorithms is given in [45]. The algorithm that is built into the composer is an extended version of enforced hill-climbing as presented in [44]. The principle of this algorithm will be briefly described in the following:

Enforced hill-climbing is a heuristic forward search algorithm in state space. Guided by a goal distance heuristic, it starts with the initial state and consecutively selects new services and reaches new states through the invocation of selected services until the goal state is reached. In a state, the algorithm will, first, determine all invocable services. This discovery task is similar to the method `findInvocableServices` from above, but does not incorporate a ranking of the invocable services. From these invocable services new states are calculated using virtual invocation: Only the post-conditions of the services are applied without actually invoking the service, leading to new, virtual states. For these states, goal distance estimations are calculated using a heuristic. The first state with a lower goal distance estimation than the current state is, then, selected as the new current state. If this state satisfies the goal state, we have found a valid composition. Otherwise, we continue by determining the now invocable services and the states reachable through their invocation until we have reached the goal state. In [44], Meyer and Weske extended this algorithm to deal with uncertainty and to compose parallel control flows.

8. Introducing Semi-Automated Composition at SAP

The purpose of this chapter is to outline how the semi-automated composition approach presented in this thesis can be introduced at SAP, using their tools and service repository. We will discuss a methodology SAP could employ to create semantic descriptions for the services in the Enterprise Service Repository. The methodology presented here departs from the process that SAP, currently, has in place to define the services in the ESR. This process will, therefore, first be described. Afterwards, we will discuss how this process can be adapted so that semantic specifications of the services in the ESR are also produced as part of the process.

The so-called PIC Governance Process is depicted in figure 8.1, using the Business Process Modeling Notation [69]. The roles participating in the process are the 'interface definition team' and the 'process integration council' (PIC). The interface definition team consists of participants from various backgrounds. SAP developers and solution managers are part of it as well as external coaches and customers. This is due to the fact that SAP concurrently pursues two ways of identifying potential services for their platform. One way is that solution managers identify the functionality that should be supported by the software. A solution manager's function is to assess which SAP products best fit a customer's needs in his specific business, and they are therefore able to identify functionality required by customers on the level of business process steps. The other way of identifying enterprise services is through a community process that SAP has set up with certain customers, independent software vendors (ISVs) and standardization bodies. The reason lies in SAP's interest in providing a set of enterprise services that is as complete as possible. Additionally, the community process also enables the aforementioned stakeholders to develop their own services for enactment on the SAP platform, thus, increasing the potential of business process differentiation.

When a solution manager or the community suggests functionality for the SAP platform, the PIC process begins. The functionality suggested at this stage is on the level of compound services, i.e. services providing an added value from the perspective of

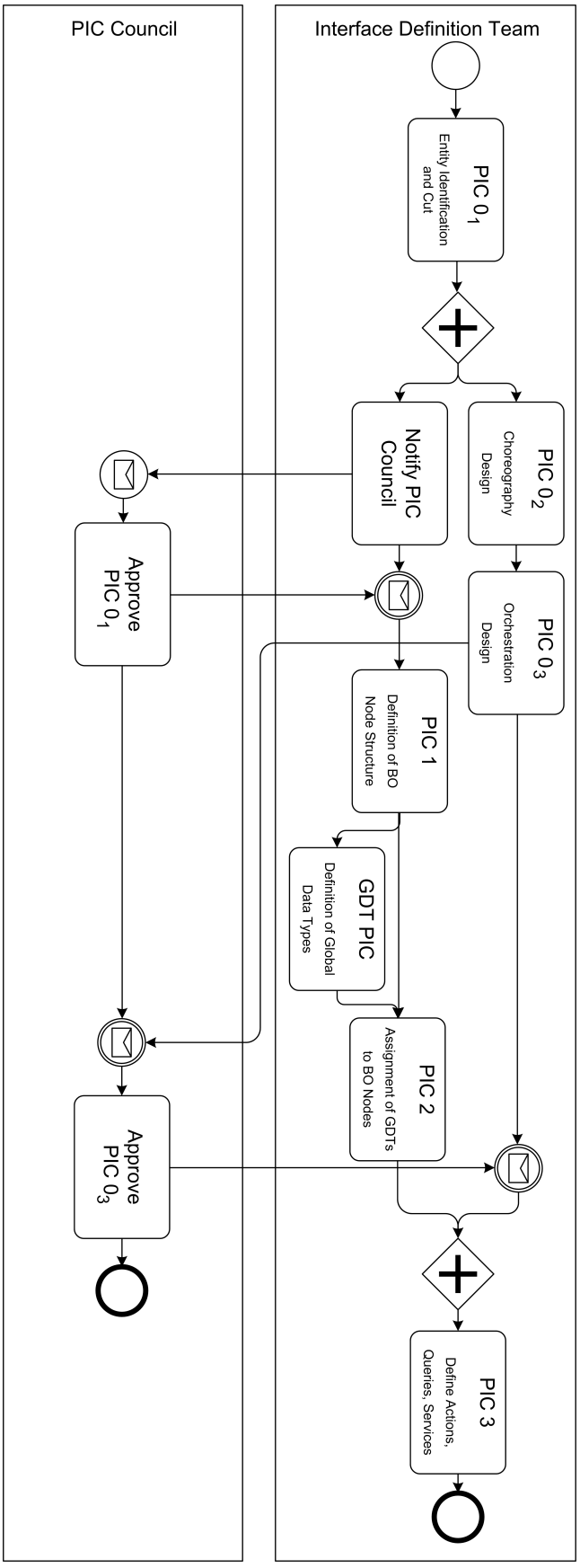

Figure 8.1: The PIC Governance Process

a business analyst. The first step, as depicted in figure 8.1, is the identification of the high-level process components required. Process components are sets of business objects that are grouped around a specific topic, such as 'Time and Leave Management', for example. A textual description of the requested functionality together with the list of involved process components, deployment units, i.e. sets of process components representing the larger system components such as 'accounting', and involved business objects are passed to the PIC council. The PIC council is an SAP-wide governance body that has an overview of the available functionality as well as the pending requests for functionality. The PIC council will stop the process if the suggested functionality is redundant. While the PIC council decides on whether to approve the PIC 0_1 stage of the request, the interface definition team carries out the steps PIC 0_2 and PIC 0_3. The outcome of PIC 0_2 is the description of an integration scenario, showing which process components and deployment units interact with each other on a higher-level. It can be seen as the choreography of the process components and deployment units involved in the business process realizing the requested functionality. PIC 0_3, then, results in the creation of a process component interaction model, showing exactly what message types are used for communication among the process components. It can be seen as the orchestration of the business process that corresponds to the functionality requested. The deliverables of PIC 0_2 and PIC 0_3 are, then, reviewed by the PIC council and are subject to approval. When the PIC council approves the PIC 0_1 stage of a request, the PIC 1 activity is triggered. In PIC 1, the business objects contained in the process components are designed if they do not already exist. This is done by specifying the business object nodes, i.e. the attributes of the business object. In PIC 2 the business object nodes are assigned to data types in the Global Data Type[1] (GDT) catalog. The GDT catalog contains all the data structures and basic data types that are allowed for use with business object nodes. In case there is no element in the GDT catalog which can be assigned to a business object node defined in PIC 1, the GDT catalog has to be extended. This step is called GDT PIC. After the completion of PIC 2 and the approval of PIC 0_3, the final step of the governance process, PIC 3, can be carried out. In this step, the actions and queries for the new business objects are designed according to the interface patterns for business objects. Furthermore, the interface of the compound service realizing the requested functionality is defined. Afterwards, the deliverables of all steps of the PIC process are stored in the ESR.

As a basis for semi-automated service composition, semantic descriptions of the services that are to be composed must be available as well as an ontology that specifies the concepts used in the semantic descriptions. In order to obtain these artifacts, the methodology described in [39] can be applied. In order to do so, the requirements for the used methodology have to be fulfilled. These requirements are that there is sufficient guidance for the design decisions on service granularity, the provision of a concise methodology to support collaboration from stakeholders from different domains, and the provision of traceability from business models to domain ontologies. The PIC Governance Process, described above, meets all those requirements. Therefore, the methodology presented in [39] can be applied. The

[1] See also section 2.2.3

methodology consists of four high-level steps, which will be described in the following. It will also be described how these high-level steps could be implemented in the context of Enterprise SOA.

1. **Identify Service Compositions** The first step of the applied methodology for the creation of semantic descriptions is to identify the 'service compositions' that are part of the application under development. The methodology assumes that a so-called 'semantic service application' is the goal of the process of specifying the domain knowledge and the semantic descriptions. Those applications are realized using various service compositions, for which dynamic bindings are, then, looked up at runtime. Since we do not aim at creating a semantic service application in this sense, we can simplify this step. The identification of service composition is in our case, essentially, the identification of the process components participating in a process as well as the process component interaction model. These artifacts are available after the step PIC 0_3 of the governance process.

2. **Design Domain Ontology** The second step of the methodology is the creation of an ontology capturing all the domain knowledge which is required for semantically specifying the services. In the context of SAP, the GDT catalog, which was referred to above, is the central location where the data types used in the service interfaces are kept. Building an ontology for the GDT catalog is, however, a challenge. In its current version, it contains 984 data types. Creating a consistent ontology that captures all of those elements, while keeping track of all possible relations among the data types, is cumbersome. Yet, it is not necessary to model all the basic types which are described in the GDT. When looking at the service interfaces of the business objects, we will see that the data types are not used in an isolated fashion. In contrast, only aggregates of elements in the GDT catalog are used. As the same aggregate structures are used for all the business objects, it is sufficient to model the ontology on the granularity of these aggregates. In doing so, the number of concepts in the ontology is drastically reduced. The ontology should, initially, be created on the basis of all the services which are PIC 0_3-approved at the point where the ontology is to be built. Then, incremental changes to the ontology should only occur at one single point in the PIC process, i.e. during the GDT PIC.

3. **Service Landscaping** The third step of the methodology used is the creation of the semantic specifications of the services that are part of a semantic service application. In our case, semantic specifications of the service operations provided by the business objects are to be created. As a prerequisite for this step, it must be known what aggregates of the elements in the GDT catalog are used for the business objects under development and what the interface of the services provided by these business objects look like. The semantic specifications can, therefore, not be created before PIC 3.

4. **Derive Semantic Query Templates** This step of the methodology is specific to the semantic service applications-based approach that is presented by the

authors of the methodology. In our case, this step is not applicable, as we do not aim to create applications in this sense, and it is, therefore, to be omitted.

The first step of the methodology does not require additional know how and the artifacts that result from this step are produced anyway during the PIC process as it is in place today at SAP. In order to carry out the second and the third steps, i.e. building the GDT ontology and the semantic service descriptions, the skill set of the interface definition team has to be extended. For the ontology and the service description used in the semi-automated composition approach described in this thesis, the Web Services Modeling Ontology (WSMO) [54] framework has been used. Therefore, people who are trained how to create the necessary specifications in WSMO by using logical formulae are required. Since the presented semi-automated composition approach is restricted in the way that it does not take assumptions and effects into account, the effort required for the creation of the ontology and the semantic descriptions is dramatically reduced. The usage of aggregate structures as a basis for the concepts in the ontology, as it was described above, further alleviates the creation of the ontology. Therefore, a total number of between two and five 'ontology engineers', i.e. persons trained to create WSMO specifications, would be sufficient in the context of SAP.

9. Conclusion

In this thesis, we have presented a novel methodology for the modeling of business processes using semi-automated composition of Web services. As a conceptual foundation, three mixed initiative features for semi-automated composition have been introduced, based on a formal model for service composition. We see these features as the functionality that is characteristic for semi-automated composition. The mixed initiative features for semi-automated composition are

- Filter Inappropriate Services, suggesting a number of relevant Web services to the modeler at every step in the process of creating the composition,
- Check Validity, summarizing the problems that would prevent the composed service from being invocable, and
- Suggest Partial Plans, inserting composed services into the composition at suitable places.

The features have been derived from a scenario taken from a recent SAP product.

The scenario is described in chapter 4. When we formally defined the mixed initiative features in chapter 5, we outlined the basic functionality to be covered by these features, as well as extensions offering more advanced functionality. On the basis of these mixed initiative features we were, then, able to compare the presented approach to related efforts in the field of semi-automated composition. One result of this evaluation, which is presented in chapter 6, is that our work is the only semi-automated composition approach that is complete in terms of functionality, i.e. it supports all three mixed initiative features. All the presented related approaches in the field of semi-automated composition are proof-of-concept implementations for very specific use cases. None of them could be extended in such a way that they would support all three mixed initiative features. The approach developed in this thesis has a broader scope and was designed to holistically cover the entire functionality characteristic for semi-automated composition. From a usability point of view, our

prototypical implementation has clear advantages over the related works. We do not impose any algorithmic planning strategy on the user. Instead, we provide a graphical user interface in the form of a BPMN-based modeler for processes. The related works either provide only a textual interface or force the user to adhere to algorithmic planning strategies, such as backward chaining. Furthermore, none of the related approaches offers control constructs for both choice and parallelism for the modeling of service compositions. The approach presented here currently provides choice, parallelism and loops, along with a clearly defined semantic of how these constructs affect the proposed mixed initiative functionality. The ontology format underlying a semi-automated composition approach is also important when we acknowledge that the formal specification of domain knowledge is a very delicate task. It might be necessary for organizations utilizing this kind of technology to re-use ontologies that have been created by other parties. Our approach is based on ontologies and semantic service descriptions in WSMO [54], an open format which was recently submitted to the W3C for standardization. We were, therefore, able to realize our prototypical implementation on top of a WSML reasoner and expect to benefit from future advances in the field of reasoning on WSML specifications. The choice of WSMO as the language used for specifying the ontology and the semantic descriptions is based on the evaluation of respective languages in chapter 3. To realize the three mixed initiative features, we have developed algorithms on top of the reasoning environment. These were described in chapter 7. Finally, we have outlined how the presented approach can be deployed at SAP. In chapter 8 we, therefore, described how the current process of developing the so-called enterprise services must be extended in order to obtain the necessary semantic descriptions.

9.1 Contributions

The most important contributions of this work are the three mixed initiative features, which we have derived from an industry scenario and, then, formally defined. Furthermore, we have shown that it is possible to create a holistic semi-automated modeling approach that supports all three mixed initiative features. We proposed a number of algorithms that can be used to realize the mixed initiative features on top of a reasoner. The goals that have been set out in section 1.2 have, therefore, been reached.

So far, two scientific publications resulted from the work on this thesis: In [55], we give an overview of current research findings in semi-automated service composition, leading to the introduction of the mixed initiative features. In [56], we present the central concepts of our realization of a service orchestration tool that supports all three mixed initiative features.

9.2 Future Work

The prototypical implementation of the approach presented here supports, at the moment, only reasoning on pre- and postconditions of Web services. Assumptions and effects have, due to limitations of the underlying WSML reasoner in its current

version, not been taken into account. While reasoning on assumptions and effects in semi-automated environments is an open point and subject to further research activities in this field, we were able to make helpful suggestions to the user only using pre- and postconditions. It has to be verified whether the additional modeling efforts that are necessary to correctly specify assumptions and effects, as described in section 4.2, are justified. It might as well be the case that the evaluation of assumptions and effects in the reasoning process causes the semi-automated composition environment to present fewer suggestions to the user, thereby, eliminating helpful hints on possible next steps.

We enhanced our prototypical modeling tool with basic performance optimizations, such as using asynchronous calls for complex queries to the backend to ensure a non-blocking system behavior. However, our prototypical implementation could be improved from a performance point of view. Optimization strategies for the semi-automated composition process are, therefore, subject to further research. We will investigate the possibilities to partition the ontology into multiple disjoint parts in order to improve the response time of the system. In doing so, the reasoning on the concepts in the ontology could be, accordingly, distributed across multiple CPUs. Furthermore, we see a potential for performance improvements regarding the algorithms that realize the mixed initiative features. Most of these algorithms, particularly those traversing the composition graph, are currently realized using recursion. As part of our future research activities, we will investigate to what extent it is possible to find iterative algorithms solving these problems.

Another direction for future work is to identify quality metrics for the usefulness of the suggestions that our system presents to the user. These metrics could, e.g. for the Filter Inappropriate Services feature, depend on the number of service operations in a given repository, as well as on specific queries and the number of suggested operations for each query. The queries are, in this context, a given state of the composition, as it is outlined in the modeling tool at a given point in time, along with one of the methods realizing the mixed initiative features, described in chapter 7. Another interesting experiment would be to deploy incomplete semantic descriptions of the services in the repository in order to measure how this affects the quality of the suggestions.

To increase the value of the presented semi-automated composition approach from a modeler's perspective, it would be conceivable to go further than only suggesting individual service operations for inclusion in the process. In the case of SAP's service repository, for example, many operations are semantically coupled with others in the sense that one operation is dependent on the other operation in order to be invocable. To optimally assist the user in modeling service compositions, the system could, therefore, suggest groups of services that are preconfigured with control flow dependencies. This can be seen as an extension of both the Filter Inappropriate Services and the Suggest Partial Plans mixed initiative feature. To improve the suggestion on the level of the individual operations, it might also be useful to track which operations the users select when creating a composition. The collected data can, then, be used to improve the service suggestions. When a modeler selects an operation A, for example, the system could suggest additional operations that other

users have also selected after they selected A. The same technique could be applied to improve the suggestion of groups of operations, as described above.

Another goal of our future research activities is to investigate how the scope of semi-automated composition can be broadened. We will aim to identify areas other than service composition to which a semi-automated approach could be applied. Such an area could, for example, be the semi-automated modeling of organizational structures, which is also an important functionality of Business Process Management Systems.

Our approach is targeted to support modelers of business processes at design time. The execution of the service compositions was not covered in this work. An open issue, therefore, remains to explore the different possibilities to ground our WSMO specifications to WSDL interfaces. Our approach could, then, be integrated into the next version of SAP's Visual Composer, which will be able to model service compositions and to deploy them to the NetWeaver platform.

The validation of our approach is currently based on our prototypical implementation demonstrating its applicability in the software industry. While we have shown that we are able to obtain reasonable support in the modeling of service compositions using the service repository and a typical SAP business process, validation with end-user is still an open point. It was planned to evaluate the modeling tool together with some of SAP's smaller partners. However, we have not yet been able to conduct user-interviews to further validate our approach.

A. Leave Request Scenario Specification

A.1 Leave Request Domain Ontology

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
namespace { _"http://www.schaffner.cc/GDTLeaveReqDomain#",
  dO _"http://www.schaffner.cc/GDTLeaveReqDomain#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax#" }

ontology _"http://www.schaffner.cc/GDTLeaveReqDomain.wsml"
  nonFunctionalProperties
    dc#title hasValue "Leave Request Scenario Ontology"
    dc#language hasValue "en-US"
    dc#contributor hasValue "Jan Schaffner"
    dc#format hasValue "text/plain"
    dc#date hasValue _date(2006,6,23)
  endNonFunctionalProperties


/* Concepts                    */
/* --------------------------- */
concept LeaveConfig

concept LeaveType
  ltype ofType _string

concept LeaveTypeBag
  membersLTB ofType (1 *) LeaveType

concept Result
  result ofType _string

concept TimePoint
```

```
31   TypeCode ofType (1 1) _integer
     Date ofType (0 1) _date
33   Time ofType (0 1) _time

35 concept TimePointPeriod
     start impliesType (1 1) TimePoint
37   end impliesType (1 1) TimePoint

39 concept LeaveRequest subConceptOf LeaveRequests
     LeaveRequestID ofType (1 1) _integer
41   LeavePeriod ofType (1 1) TimePointPeriod

43 concept Person
     FirstName ofType _string
45   LastName ofType _string

47 concept Employee subConceptOf Person
     EmployeeID ofType (1 1) _integer
49
   concept Manager subConceptOf Employee
51
   concept EmployeeBag
53   membersEMP ofType (1 *) Employee

55 concept TimeAccountType
     tatype ofType _string
57
   concept EmployeeTimeAccount subConceptOf EmployeeTimeBalance
59   Balance ofType (1 1) _integer

61 concept LeaveRequestState
     state ofType _string
63
   concept EmployeeTimeBalance
65   membersETA ofType (1 *) EmployeeTimeAccount

67 concept LeaveRequestBag
     membersLRQ ofType (1 *) LeaveRequest
69

71 /* Relations                    */
   /* ------------------------------ */
73 relation hasRequestor(ofType LeaveRequest, ofType Employee)

75 relation hasConfig(ofType LeaveRequest, ofType LeaveConfig)

77 relation hasLeaveType(ofType LeaveConfig, ofType LeaveType)

79 relation hasPeriod(ofType LeaveRequest, ofType TimePointPeriod)
```

```
81 relation hasTimeAccountType(ofType EmployeeTimeAccount,
     ofType TimeAccountType)
83
   relation hasLeaveRequestState(ofType LeaveRequest,
85   ofType LeaveRequestState)

87 relation hasEmployeeTimeBalance(ofType Employee,
     ofType EmployeeTimeBalance)
89
   relation hasManager(ofType Employee, ofType Manager)
91
   relation hasLeaveRequest(ofType Employee, ofType LeaveRequest)
93

95 /* Instances                    */
   /* ---------------------------- */
97 instance fulldayType memberOf LeaveType
     ltype hasValue "fullday"
99
   instance halfdayType memberOf LeaveType
101  ltype hasValue "halfday"

103 instance illnesswithcertificateType memberOf LeaveType
     ltype hasValue "illnesswithcertificate"
105
   instance illnesswithoutcertificateType memberOf LeaveType
107  ltype hasValue "illnesswithoutcertificate"

109 instance requestedState memberOf LeaveRequestState
     state hasValue "requested"
111
   instance approvedState memberOf LeaveRequestState
113  state hasValue "approved"

115 instance declinedState memberOf LeaveRequestState
     state hasValue "declined"
117
   instance okResult memberOf Result
119   result hasValue "ok"

121 instance failedResult memberOf Result
     result hasValue "failed"
123
   instance paidvacationType memberOf TimeAccountType
125  tatype hasValue "paidvacation"

127 instance overtimeType memberOf TimeAccountType
     tatype hasValue "overtime"
```

```
129
instance sickleaveType memberOf TimeAccountType
131   tatype hasValue "sickleave"
```

A.2 Leave Request Enterprise Service Operations

In the following, the WSMO `webService` specifications of the service operations used in the leave request scenario are listed.

A.2.1 Read Leave Request Configuration by Employee

```
1  webService _'http://localhost:1080/LeaveRequest/
               ReadLeaveRequestConfigurationByEmployee.wsml'
3    importsOntology _'http://localhost:1080/LeaveRequest/
                       GDTLeaveReqDomain.wsml'
5      capability EmpLeaReqConByEmpCapability
       nfp
7        dO#Role hasValue 'EmployeeRole'
         dO#EnterpriseService hasValue 'Time and Leave Management'
9      endnfp
       sharedVariables {?conf, ?ltype, ?members}
11     precondition
         nfp
13         dc#description hasValue 'Takes Employee, date as input.'
         endnfp
15       definedBy
           ?emp memberOf dO#Employee
17           and ?date memberOf wsml#date.
       postcondition
19       nfp
           dc#description hasValue 'Produces a LeaveConfig as
21           well as contained data structures as output.'
         endnfp
23       definedBy
           ?conf memberOf dO#LeaveConfig
25           and ?allowedTypes memberOf dO#LeaveTypeBag
             and ?allowedTypes[dO#membersLTB hasValue ?members].
27     effect
         definedBy
29         forall ?ltype (?ltype memberOf ?members implies
             dO#hasLeaveType(?conf,?ltype)).
```

A.2.2 Read Employee Time Account

```
webService _'http://localhost:1080/LeaveRequest/ReadEmployeeTimeAccount.wsml'
2    importsOntology _'http://localhost:1080/LeaveRequest/
                       GDTLeaveReqDomain.wsml'
4    capability EmpTimByEmpCapability
       nfp
```

```
    dO#Role hasValue 'EmployeeRole'
    dO#EnterpriseService hasValue 'Time and Leave Management'
  endnfp
  sharedVariables {?emp, ?bal, ?pvTimAcc, ?ovTimAcc, ?slTimAcc}
  precondition
    nfp
      dc#description hasValue 'Takes Employee, date as input.'
    endnfp
    definedBy
      ?emp memberOf dO#Employee
        and ?date memberOf wsml#date.
  postcondition
    nfp
      dc#description hasValue 'Produces EmployeeTimeBalance as
        well as contained data structures as output.'
    endnfp
    describedBy
      ?bal memberOf dO#EmployeeTimeBalance

        and ?pvTimAcc memberOf dO#EmployeeTimeAccount
        and ?pvTimAccTyp[dO#type hasValue 'paid-vacation']
          memberOf dO#TimeAccountType
        and ?bal[dO#membersETA hasValue ?pvTimAcc]

        and ?otTimAcc memberOf dO#EmployeeTimeAccount
        and ?otTimAccTyp[dO#type hasValue 'overtime']
          memberOf dO#TimeAccountType
        and ?bal[dO#membersETA hasValue ?otTimAcc]

        and ?slTimAcc memberOf dO#EmployeeTimeAccount
        and ?slTimAccTyp[dO#type hasValue 'sick-leave']
          memberOf dO#TimeAccountType
        and ?bal[dO#membersETA hasValue ?slTimAcc].
  effect
    definedBy
        and dO#hasTimeAccountType(?pvTimAcc, ?pvTimAccTyp)
        and dO#hasTimeAccountType(?otTimAcc, ?otTimAccTyp)
        and dO#hasTimeAccountType(?slTimAcc, ?slTimAccTyp)
        and dO#hasEmployeeTimeBalance(?emp, ?bal).
```

A.2.3 Find Leave Request by Employee

```
webService _'http://localhost:1080/LeaveRequest/
              FindLeaveRequestByEmployee.wsml'
  importsOntology _'http://localhost:1080/LeaveRequest/
                      GDTLeaveReqDomain.wsml'
  capability EmpLeaReqByEmpCapability
    nfp
      dO#Role hasValue 'ManagerRole'
      dO#EnterpriseService hasValue 'Time and Leave Management'
```

```
    endnfp
    sharedVariables {?emp, ?lrq, ?lrqs, ?state}
    precondition
      nfp
        dc#description hasValue 'Takes an Employee as input.'
      endnfp
      definedBy
        ?emp memberOf dO#Employee.
    postcondition
      nfp
        dc#description hasValue 'Produces a LeaveRequestBag as
          output. Contains all LeaveRequests for Employee that
          are in requestedState.'
      endnfp
      definedBy
        ?lreqbag memberOf dO#LeaveRequestBag
          and ?lreqbag[membersLRQ hasValue ?lrqs]
          and ?state memberOf dO#LeaveRequestState
          and ?state[state hasValue 'requested'].
    effect
      definedBy
          forall ?lrq (?lrq memberOf ?lrqs implies
            ( dO#hasLeaveRequest(?emp, ?lrq) and
              dO#hasLeaveRequestState(?lrq, ?state))
            ).
```

A.2.4 Find Leave Request Allowed Approver by Employee

```
webService _'http://localhost:1080/LeaveRequest/
            FindLeaveRequestAllowedApproverByEmployee.wsml'
  importsOntology _'http://localhost:1080/LeaveRequest/
                    GDTLeaveReqDomain.wsml'
  capability EmpLeaReqAllAppByEmpCapability
    nfp
      dO#Role hasValue 'EmployeeRole'
      dO#EnterpriseService hasValue 'Time and Leave Management'
    endnfp
    sharedVariables {?emp, ?mgr}
    precondition
      nfp
        dc#description hasValue 'Takes Employee as input.'
      endnfp
      definedBy
        ?emp memberOf dO#Employee.
    postcondition
      nfp
        dc#description hasValue 'Produces Manager as output.'
      endnfp
      definedBy
        ?mgr memberOf dO#Manager
```

```
    effect
      definedBy
        dO#hasManager(?emp,?mgr).
```

A.2.5 Check Create Leave Request

```
webService _'http://localhost:1080/LeaveRequest/CheckCreateLeaveRequest.wsml'
  importsOntology _'http://localhost:1080/LeaveRequest/
                    GDTLeaveReqDomain.wsml'
  capability EmpLeaReqCheCapability
    nfp
      dO#Role hasValue 'EmployeeRole'
      dO#EnterpriseService hasValue 'Time and Leave Management'
    endnfp
    precondition
      nfp
        dc#description hasValue 'Takes Employee, TimePointPeriod,
          LeaveType as input.'
      endnfp
      definedBy
        ?emp memberOf dO#Employee
      and ?lperd memberOf dO#TimePointPeriod
          and ?ltype memberOf dO#LeaveType.
    postcondition
      nfp
        dc#description hasValue 'Produces a Result with value
          'ok' as output.'
      endnfp
      definedBy
        ?res memberOf dO#CheckCreateLeaveRequestResult
          and ?res[dO#result hasValue 'ok'].
```

A.2.6 Create Leave Request

```
webService _'http://localhost:1080/LeaveRequest/CreateLeaveRequest.wsml'
  importsOntology _'http://localhost:1080/LeaveRequest/
                    GDTLeaveReqDomain.wsml'
  capability EmpLeaReqCapability
    nfp
      dO#Role hasValue 'EmployeeRole'
      dO#EnterpriseService hasValue 'Time and Leave Management'
    endnfp
    sharedVariables {?lperd,?ltype,?emp,?lrq}
    precondition
      nfp
        dc#description hasValue 'Takes Employee, TimePointPeriod,
          LeaveType as input.'
        endnfp
      definedBy
    ?emp memberOf dO#Employee
```

```
17      and ?lperd memberOf dO#TimePointPeriod
        and ?ltype memberOf dO#LeaveType
19      and ?result memberOf dO#CheckCreateLeaveRequestResult.
    postcondition
21      nfp
          dc#description hasValue 'Produces LeaveRequest as well as
23        contained data structures as output. The newly created
          LeaveRequest is set to requestedState.'
25      endnfp
        definedBy
27        ?lrq memberOf dO#LeaveRequest
            and ?lrq[dO#LeavePeriod hasValue ?lperd]
29          and ?lrqstate memberOf dO#LeaveRequestCreatedState
            and ?lrqstate[state hasValue 'requested'].
31  effect
      definedBy
33        and dO#hasConfig(?lrq,?conf)
          and dO#hasType(?conf,?ltype)
35        and dO#hasLeaveRequest(?emp,?lrq)
          and dO#hasLeaveRequestState(?lrq,?lrqstate).
```

A.2.7 Find Leave Request by ID

```
webService _'http://localhost:1080/LeaveRequest/FindLeaveRequestById.wsml'
2  importsOntology _'http://localhost:1080/LeaveRequest/
                       GDTLeaveReqDomain.wsml'
4  capability EmpLeaReqByIdCapability
     nfp
6      dO#Role hasValue 'ManagerRole'
       dO#EnterpriseService hasValue 'Time and Leave Management'
8    endnfp
     sharedVariables {?lrqid}
10   precondition
       nfp
12       dc#description hasValue 'Takes LeaveRequestID as input.'
       endnfp
14     definedBy
         ?lrqid memberOf dO#LeaveRequestID.
16   postcondition
       nfp
18       dc#description hasValue 'Produces a LeaveRequest
           as output.'
20     endnfp
       definedBy
22       ?lrq memberOf dO#LeaveRequest
         and ?lrq[LeaveRequestID hasValue ?lrqid].
```

A.2.8 Find Reporting Employee by Employee

```
1 webService _'http://localhost:1080/LeaveRequest/
```

```
                FindReportingEmployeeByEmployee.wsml'
  importsOntology _'http://localhost:1080/LeaveRequest/
                    GDTLeaveReqDomain.wsml'
  capability RepEmpByEmpCapability
    nfp
      dO#Role hasValue 'ManagerRole'
      dO#EnterpriseService hasValue 'Time and Leave Management'
    endnfp
    sharedVariables {?mgr, ?emp, ?bagmmbers}
    precondition
      nfp
        dc#description hasValue 'Takes Manager as input.'
      endnfp
      definedBy
        ?mgr memberOf dO#Manager.
    postcondition
      definedBy
        ?empbag memberOf dO#EmployeeBag
          and ?empbag[dO#membersEMP hasValue ?bagmembers].
    effect
      nfp
        dc#description hasValue 'Produces a bag of Employees
          as output.'
      endnfp
      definedBy
        forall ?emp (?emp memberOf ?bagmembers
          implies dO#hasManager(?emp,?mgr) ).
```

A.2.9 Check Approve Leave Request

```
webService _'http://localhost:1080/LeaveRequest/CheckApproveLeaveRequest.wsml'
  importsOntology _'http://localhost:1080/LeaveRequest/
                    GDTLeaveReqDomain.wsml'
  capability EmpLeaReqAppCheCapability
    nfp
      dO#Role hasValue 'ManagerRole'
      dO#EnterpriseService hasValue 'Time and Leave Management'
    endnfp
    sharedVariables {?lrq, ?conf, ?ltype}
    precondition
      nfp
        dc#description hasValue 'Takes LeaveRequest as input.'
      endnfp
      definedBy
        ?lrq memberOf dO#LeaveRequest
          and ?conf memberOf dO#LeaveRequestConfig
          and ?state memberOf LeaveRequestCreatedState
          and ?state[state hasValue 'requested']
          and ?ltype memberOf dO#LeaveType
          and ( ?ltype[ltype hasValue 'halfdayType']
```

```
        or ?ltype[ltype hasValue 'fulldayType'] ).
  assumption
    definedBy
      dO#hasConfig(?lrq,?conf)
        and dO#hasLeaveRequestState(?lrq,?state)
        and dO#hasLeaveType(?conf, ?ltype).
  postcondition
    nfp
      dc#description hasValue 'Produces a Result with value
        'ok' as output.'
    endnfp
    definedBy
      ?res memberOf dO#CheckApproveLeaveRequestResult
      and ?res[dO#result hasValue 'ok'].
```

A.2.10 Approve Leave Request

```
webService _'http://localhost:1080/LeaveRequest/ApproveLeaveRequest.wsml'
  importsOntology _'http://localhost:1080/LeaveRequest/
                    GDTLeaveReqDomain.wsml'
  capability EmpLeaReqAppCapability
    nfp
      dO#Role hasValue 'ManagerRole'
      dO#EnterpriseService hasValue 'Time and Leave Management'
    endnfp
    sharedVariables {?lrq, ?conf, ?state}
    precondition
      nfp
        dc#description hasValue 'Takes LeaveRequest as input. LeaveRequest
          must be in requestedState. LeaveRequestType must be halfdayType,
          fulldayType. Sick leaves do not require approval.'
      endnfp
      definedBy
        ?lrq memberOf dO#LeaveRequest
          and ?conf memberOf dO#LeaveRequestConfig
          and ?res memberOf dO#CheckApproveLeaveRequestResult
          and ?state memberOf LeaveRequestCreatedState
          and ?state[state hasValue 'requested']
          and ?ltype memberOf dO#LeaveType
          and ( ?ltype[ltype hasValue 'halfdayType']
            or ?ltype[ltype hasValue 'fulldayType'] ).
    assumption
      definedBy
        dO#hasConfig(?lrq,?conf)
        and dO#hasLeaveRequestState(?lrq,?state)
        and dO#hasLeaveType(?conf, ?ltype).
    postcondition
      nfp
        dc#description hasValue 'After the execution the LeaveRequest
          is in approvedState.'
```

```
34      endnfp
        definedBy
36        ?lrqstate memberOf dO#LeaveRequestState
            and dO#hasLeaveRequestApprovedState(?lrq,?lrqstate)
38          and ?lrqstate[state hasValue 'approved'].
```

A.2.11 Check Reject Leave Request

```
webService _'http://localhost:1080/LeaveRequest/CheckRejectLeaveRequest.wsml'
2  importsOntology _'http://localhost:1080/LeaveRequest/
                       GDTLeaveReqDomain.wsml'
4  capability EmpLeaReqRejCheCapability
     nfp
6      dO#Role hasValue 'ManagerRole'
       dO#EnterpriseService hasValue 'Time and Leave Management'
8    endnfp
     sharedVariables {?lrq, ?conf, ?state, ?ltype}
10   precondition
       nfp
12       dc#description hasValue 'Takes LeaveRequest as input.'
       endnfp
14     definedBy
         ?lrq memberOf dO#LeaveRequest
16         and ?conf memberOf dO#LeaveRequestConfig
           and ?state memberOf LeaveRequestCreatedState
18         and ?state[state hasValue 'requestedState']
           and ?ltype memberOf dO#LeaveType
20         and ( ?ltype[ltype hasValue 'halfdayType']
             or ?ltype[ltype hasValue 'fulldayType'] ).
22   assumption
       definedBy
24       dO#hasConfig(?lrq, ?conf)
           and dO#hasLeaveRequestState(?lrq, ?state)
26         and dO#hasLeaveType(?conf, ?ltype).
     postcondition
28     nfp
         dc#description hasValue 'Produces a Result with value 'ok'
30       as output.'
       endnfp
32     definedBy
         ?res memberOf dO#CheckRejectLeaveRequestResult
34         and ?res[dO#result hasValue 'ok'].
```

A.2.12 Reject Leave Request

```
webService _'http://localhost:1080/LeaveRequest/RejectLeaveRequest.wsml'
2  importsOntology _'http://localhost:1080/LeaveRequest/
                       GDTLeaveReqDomain.wsml'
4  capability EmpLeaReqRejCapability
     nfp
```

```
6       dO#Role hasValue 'ManagerRole'
        dO#EnterpriseService hasValue 'Time and Leave Management'
8     endnfp
      sharedVariables {?lrq, ?conf, ?state, ?ltype}
10    precondition
        nfp
12        dc#description hasValue 'Takes LeaveRequest as input.'
        endnfp
14      definedBy
          ?lrq memberOf dO#LeaveRequest
16          and ?conf memberOf dO#LeaveRequestConfig
            and ?res memberOf dO#CheckRejectLeaveRequestResult
18          and ?state memberOf LeaveRequestCreatedState
            and ?state[state hasValue 'requestedState']
20          and ?ltype memberOf dO#LeaveType
            and ( ?ltype[ltype hasValue 'halfdayType']
22            or ?ltype[ltype hasValue 'fulldayType'] ).
      assumption
24      definedBy
          dO#hasConfig(?lrq,?conf)
26          and dO#hasLeaveRequestState(?lrq, ?state)
            and dO#hasLeaveType(?conf, ?ltype).
28    postcondition

30    effect
        nfp
32        dc#description hasValue 'After the execution the
            LeaveRequest is in declinedState.'
34      endnfp
        definedBy
36        ?lrqstate memberOf dO#LeaveRequestRejectedState
            and dO#hasLeaveRequestState(?lrq, ?lrqstate)
38          and ?lrqstate[state hasValue 'declined'].
```

Bibliography

[1] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A Service Creation Environment Based on End to End Composition of Web Services. In *WWW '05: Proceedings of the 14th Cnternational Conference on World Wide Web*, pages 128–137, New York, NY, USA, 2005. ACM Press.

[2] R. Akkiraju et al. Semantic Tools for Web Services, 2005. `http://www.alphaworks.ibm.com/tech/wssem`.

[3] R. Akkiraju et al. Web Service Semantics - WSDL-S. Technical report, LSDIS and the University of Georgia, Apr 2005. `http://lsdis.cs.uga.edu/projects/METEOR-S/WSDL-S/`.

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004.

[5] A. Arkin et al. Web Service Choreography Interface (WSCI) 1.0. Technical report, World Wide Web Consortium (W3C), Aug 2002. `http://www.w3.org/TR/2002/NOTE-wsci-20020808`.

[6] A. Arkin et al. XML Schema Part 0: Primer Second Edition. Technical report, World Wide Web Consortium (W3C), Oct 2004. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`.

[7] S. Battle et al. Semantic Web Services Framework (SWSF) Overview. Technical report, World Wide Web Consortium (W3C), Sep 2005. `http://www.w3.org/Submission/2005/SUBM-SWSF-20050909/`.

[8] D. Berardi, D. Calvanese, G. De Giacomo, and M. Mecella. Composition of Services with Nondeterministic Observable Behavior. In *Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC'05)*, volume 3826 of *Lecture Notes in Computer Science*, pages 520–526. Springer, 2005.

[9] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5), May 2001.

[10] T. Boubez, M. Hondo, C. Kurt, J. Rodriguez, and D. Rogers. UDDI Programmer's API 1.0: UDDI Published Specification. Technical report, OASIS, Jun 2002. `http://uddi.org/pubs/ProgrammersAPI-V1.01-Published-20020628.pdf`.

[11] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1. Technical report, World Wide Web Consortium (W3C), Jun 2006. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.

[12] R. Buddharaju et al. Dublin Core Metadata Element Set, Version 1.1: Reference Description. Technical report, Dublin Core Metadata Initiative, Dec 2004. http://dublincore.org/documents/2004/12/20/dces/.

[13] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, World Wide Web Consortium (W3C, Mar 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[14] E. Cimpian, M. Moran, E. Oren, T. Vitvar, and M. Zaremba. D13.0v0.2 Overview and Scope of WSMX. Technical report, The ESSI WSMO working group, Feb 2005. http://www.wsmo.org/TR/d13/d13.0/v0.2/20050208/.

[15] M. Crawford. Core Components Technical Specification. Technical report, United Nations Economic Commission for Europe, Nov 2003. http://www.unece.org/cefact/tmg/cefact_ccts_v2_01.pdf.

[16] J. de Bruijn and H. Lausen. Web Service Modeling Language (WSML). Technical report, World Wide Web Consortium (W3C), Jun 2005. http://www.w3.org/Submission/2005/SUBM-WSML-20050603/.

[17] M. Dean and G. Schreiber. OWL Web Ontology Language Reference. Technical report, World Wide Web Consortium (W3C), Feb 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[18] E. Dumbill. XML Watch: Finding Friends with XML and RDF. Technical report, IBM, Jun 2002. http://www-106.ibm.com/developerworks/xml/library/x-foaf.html.

[19] D. C. Fallside and P. Walmsley. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, Oct 2005. http://www.oasis-open.org/apps/org/workgroup/wsbpel/.

[20] J. Farrell and H. Lausen. Semantic Annotations for WSDL. Technical report, World Wide Web Consortium (W3C), Jun 2006. http://www.w3.org/2002/ws/sawsdl/spec/.

[21] T. Fiedler, H. Meinert, and V. Wiechers. ESA Architecture Series 2006: Services (Draft). XU Product Architecture Knowledge Transfer, SAP Internal / Confidential, Feb 2006.

[22] Y. Gil. Description Logics and Planning. *AI Magazine*, 26(2):62–71, 2005.

[23] M. Gilpin and J. Hoppermann. SOA Energizes Business Processes. *Tech Choices*, May 2006.

[24] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[25] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. Technical report, World Wide Web Consortium (W3C), Jun 2006. `http://www.w3.org/TR/2003/REC-soap12-part1-20030624/`.

[26] V. Haarslev and R. Möller. Racer: A Core Inference Engine for the Semantic Web. In *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003), located at the 2nd International Semantic Web Conference ISWC 2003, Sanibel Island, Florida, USA, October 20*, pages 27–36, 2003.

[27] F. Hakimpour, D. Sell, L. Cabral, J. Domingue, and E. Motta. Semantic Web Service Composition in IRS-III: The Structured Approach. In *7th IEEE International Conference on E-Commerce Technology (CEC 2005), 19-22 July 2005, München, Germany*, pages 484–487. IEEE Computer Society, 2005.

[28] G. Hench et al. WSML 2 Reasoner Framework. `http://dev1.deri.at/wsml2reasoner/`.

[29] J. Hündling and M. Weske. Modeling Quality of Services in Service Oriented Environments. In *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004)*, pages 75–80, Paphos, Cyprus, 2004. Department of Computer Science, University of Cyprus.

[30] D. Hollingsworth. The Workflow Reference Model. Technical report, The Workflow Management Coalition, Jan 1995. `http://www.wfmc.org/standards/docs/tc003v11.pdf`.

[31] M. Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer, Berlin, Germany, 2004.

[32] M. C. Jäger, G. Rojec-Goldmann, C. Liebetruth, G. Mühl, and K. Geihs. Ranked Matching for Service Descriptions Using OWL-S. In *Kommunikation in Verteilten Systemen (KiVS), 14. ITG/GI-Fachtagung Kommunikation in Verteilten Systemen (KiVS 2005) Kaiserslautern, 28. Februar - 3. März 2005*, Informatik Aktuell, pages 91–102. Springer, 2005.

[33] M. C. Jäger, G. Rojec-Goldmann, and G. Muhl. QoS Aggregation for Web Service Composition using Workflow Patterns. In *EDOC '04: Proceedings of the Eighth International Conference on Enterprise Distributed Object Computing*, pages 149–159, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[34] J. Kim, M. Spraragen, and Y. Gil. An Intelligent Assistant for Interactive Workflow Composition. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*, pages 125–131, New York, NY, USA, 2004. ACM Press.

[35] A. Knöpfel, B. Gröne, and P. Tabeling. *Fundamental Modeling Concepts: Effective Communication of IT Systems*. Wiley and Sons Ltd., 2004.

[36] J. Kopecký et al. D24.2v0.1. WSMO Grounding. Technical report, The ESSI WSMO working group, Jun 2005. `http://www.wsmo.org/TR/d24/d24.2/v0.1/20050611/`.

[37] D. Kuropka and M. Weske. Die Adaptive Services Grid Plattform: Motivation, Potential, Funktionsweise und Anwendungsszenarien. *EMISA Forum*, 26(1), Jan 2006.

[38] G. Laures. Are Service-oriented Architectures the Panacea for a High-Availability Challenge? In *Proceedings 2nd International Service Availability Forum (ISAS'05)*, Lecture Notes In Computer Science, Heidelberg, Germany, Apr 2005. Springer.

[39] G. Laures, H. Meyer, and M. Breest. An Engineering Method for Semantic Service Applications. In *Proceedings of the 1st International Workshop on Design of Service-Oriented Applications (WDSOA'05)*, pages 79–86, Amsterdam, The Netherlands, 2005. IBM Research Division.

[40] H. Lausen, U. Keller, and D. Anicic. MINS Reasoner. `http://dev1.deri.at/mins/`.

[41] L. Li and I. Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. In *Proceedings of the Twelfth International World Wide Web Conference (WWW2003), 20-24 May, 2003, Budapest, Hungary*. ACM, 2003.

[42] D. Martin et al. OWL-S: Semantic Markup for Web Services. Technical report, The Defense Advanced Research Projects Agency (DARPA), Nov 2003. `http://www.daml.org/services/`.

[43] H. Meyer. Development and Realization of a Planning Component for Service Composition (in German). Diplomarbeit, Institute of Computer Science, University of Potsdam, Potsdam, Germany, 2005.

[44] H. Meyer and M. Weske. Automated Service Composition using Heuristic Search. In S. Dustdar, J. L. Fiadeiro, and A. Sheth, editors, *Proceedings of the Fourth International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes In Computer Science*, pages 81–96, Heidelberg, Germany, 2006. Springer.

[45] N. Milanovic and M. Malek. Current solutions for Web service composition. *IEEE Internet Computing*, 8(6):51–59, 2003.

[46] C. Moore. The Forrester Wave^{TM}: Human-Centric Business Process Management Suites, Q1 2006. *Tech Choices*, Feb 2006.

[47] B. Motik and U. Sattler. Practical DL Reasoning over Large ABoxes with KAON2. FZI Karlsruhe, Germany, submitted for publication, 2006.

[48] K. L. Myers et al. PASSAT: A User-centric Planning Framework. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, Houston, TX, USA, 2002. AAAI.

[49] Network Working Group. Hypertext Transfer Protocol – HTTP/1.1. Technical report, The Internet Engineering Task Force (IETF), Jun 1999. `ftp://ftp.isi.edu/in-notes/rfc2616.txt`.

[50] Object Management Group (OMG). UML Superstructure Specification, v2.0. Technical report, May 2004. `http://www.omg.org/cgi-bin/doc?formal/05-07-04`.

[51] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and Monitoring Web Service Composition. *Lecture Notes in Computer Science*, 3192:106–115, Jan 2004.

[52] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proceedings of the Eleventh International World Wide Web Conference (WWW02), 7-11 May, 2002, Hawaii, USA*. ACM, 2002.

[53] J. Rao, D. Dimitrov, P. Hofmann, and N. Sadeh. A Mixed Initiative Approach to Semantic Web Service Discovery and Composition: SAP's Guided Procedures Framework. In *Proceedings of the International Conference on Web Services (ICWS'06), September 18-22, 2006, Chicago, USA*. IEEE Computer Society, 2006.

[54] D. Roman, H. Lausen, and U. Keller. D2v1.2. Web Service Modeling Ontology (WSMO). Technical report, The ESSI WSMO working group, Apr 2005. `http://www.wsmo.org/TR/d2/v1.2/20050413/`.

[55] J. Schaffner and H. Meyer. Mixed Initiative Use Cases For Semi-Automated Service Composition: A Survey. In *Proceedings of the International Workshop on Service Oriented Software Engineering (IW-SOSE'06), located at ICSE'06, Shanghai, China*. ACM Press, New York, NY, USA, May 2006.

[56] J. Schaffner, H. Meyer, and C. Tosun. A Semi-automated Orchestration Tool for Service-based Business Processes. In *Proceedings of the 2nd International Workshop on Engineering Service-Oriented Applications: Design and Composition (WESOA'06), located at ICSOC'06, Chicago, USA*, Lecture Notes In Computer Science, Heidelberg, Germany, Dec 2006. Springer. to appear.

[57] H. Schuschel and M. Weske. Automated Planning in a Service-Oriented Architecture. In *Proceedings of the 3rd International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises*, pages 75–80. IEEE Computer Society, 2004.

[58] M. Seubert et al. Data Type Catalog - Definitions of Global Data Types and Core Data Types. Interface Coaching Team, SAP Internal / Confidential, May 2006.

[59] E. Sirin, B. Parsia, and J. Hendler. Filtering and Selecting Semantic Web Services with Interactive Composition Techniques. *IEEE Intelligent Systems*, 19:42–49, 2004.

[60] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN Planning for Web Service Composition Using SHOP2. *Journal of Web Semantics*, 1(4):377–396, 2004.

[61] K. P. Sycara. Dynamic Discovery, Invocation and Composition of Semantic Web Services. In *Methods and Applications of Artificial Intelligence, Third Helenic Conference on AI, SETN 2004, Samos, Greece, May 5-8, 2004, Proceedings*, volume 3025 of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2004.

[62] X. Tan. Reasoning with SWSO Using Vampire. `http://www.cs.toronto.edu/~sheila/2542/w06/readings/xing_present.pdf`, May 2006.

[63] A. Tate. Generating Project Networks. In *Proceedings of the Fifth Joint Conference on Artificial Intelligence, Cambridge, MA, USA*, pages 888–893. Morgan Kaufmann Publishers, 1977.

[64] S. Thatte. XLANG. Technical report, Microsoft Corporation, 2001. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`.

[65] The World Wide Web Consortium. W3C Renews Web Services Activity, Expanding Work, Mar 2006. `http://www.w3.org/2006/03/saws-pressrelease`.

[66] W. van der Aalst and A. H. ter Hofstede. Verification of Workflow Nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN'97), London, UK*, Lecture Notes In Computer Science, pages 407–426, Heidelberg, Germany, 1997. Springer.

[67] W. van der Aalst, A. H. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. *Distributed Parallel Databases*, 14(1):5–51, 2003.

[68] W. van der Aalst, A. H. ter Hofstede, and M. Weske. Business Process Management: A Survey. *Lecture Notes in Computer Science*, 2678:1–12, Jan 2003.

[69] S. A. White. Business Process Modeling Notation, Working Draft (1.0). Technical report, The Business Process Modeling Initiative, Aug 2003. `http://www.bpmi.org/bpmi-downloads/BPMN-1-0_draft.zip`.

[70] G. Yang, M. Kifer, and C. Zhao. Flora-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003*, volume 2888 of *Lecture Notes in Computer Science*, pages 671–688. Springer, 2003.

[71] L. Zeng, B. Benatallah, H. Lei, A. H. H. Ngu, D. Flaxer, and H. Chang. Flexible Composition of Enterprise Web Services. *Electronic Markets*, 13(2), 2003.

Erklärung

Ich versichere hiermit, dass ich die vorliegende Masterarbeit selbständig, ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Potsdam, den 22. Dezember 2006

Jan Schaffner