HAMBURG UNIVERSITY OF TECHNOLOGY
INSTITUTE FOR SOFTWARE SYSTEMS

On The Extension and Integration of Existing SAP Enterprise Services in The Context of SOA4All

PATRICK UN

Supervisor: Prof. Dr. RALF MÖLLER
Co-supervisor: Prof. Dr. DIETER GOLLMANN
Advisor: Dr. JÜRGEN VOGEL

Submitted in conformity and partial fulfillment of the requirements for the degree of Master of Computer Science in Information and Media Technologies

**Hamburg University of Technology
Technische Universität Hamburg-Harburg**

Hamburg, August 2009

Abstract

The SOA4All project aims to provide an user-friendly service delivery platform to a wide range of users consisting of both experts and laymen. This novel platform uses semantic web technology to annotate syntactic based web services and use logical reasoning to discover and mediate a large number of services for easier consumption and efficient delivery to end users. At SAP Research, the integration of existing, currently non-semantic SAP enterprise services into the SOA4All framework is studied. In order to obtain a set of synthesized complex functionalities for a process scenario of business registration at governmental administration which is supported by automated ERP systems using existing enterprise services, we show that using a well-known and proven approach of dynamical basic action theory in the situation calculus to axiomatize the domain can soundly and correctly capture the interaction dynamics, invocation constraints of the component enterprise services so that the constraints and requirements for integration are satisfied. With an in-depth survey of a series of semantic service matching and automatic service composition approaches in the industry and academia, we provide a consolidated, critical view of contemporary intelligent agent based solutions to the service composition problem in the domain of knowledge representation and artificial intelligence. We argue that it can suitably solve the challenging integration problem of existing enterprise services.

Dedication

This work is dedicated to my parents for their unconditional and infinite love.

Acknowledgements

I owe my deep gratitude and respect to my supervisor Prof. Dr. rer.-nat. habil. Ralf Möller, professor at the Institute for Software Systems at the Hamburg University of Technology, whose wisdom, knowledge and experiences shape many insightful conversations during the course of this thesis, without which many ideas would not have been well developed or contemplated on. His patient guidance, understanding, very congenial and helpful advice at every stage of my work has given the correct direction for my research and has helped me to make this thesis possible. I owe deep and heartfelt gratitude to my co-supervisor Prof. Dr. rer.-nat. habil. Dieter Gollmann who is very kind in offering help and stimulating thought and thank him greatly for co-supervising my thesis.
I owe sincere gratitude to Dr. Michael Wessel at the Institute for Software Systems, for his patient answering of my questions and giving me clarification of many important concepts. I shall always remember his benevolence and often wittiness in our conversations.

I must also thank Prof. Wolfgang Bauhofer from my whole heart for his kindness, unconditional help and supervision of my NIT propositions. Prof. Bauhofer's fruitful guidance during the NIT studies will always remain vivid in my mind.

To my advisor at SAP Research Switzerland, Dr. Jürgen Vogel, I want to thank him from my heart for his patience, his brilliant thought and his benevolence which is intrinsic to his kind personality. He has helped me at every stage of this thesis, especially concerning the topics of SAP applications. My thank also goes to Florian Stroh who as a team member has been enormously helpful in many challenging situations and contributing to a harmonious and fun working environment.

Declaration

I declare that this thesis has been prepared by me independently,
all literal or citations are clearly indicated with appropriate source references,
and that no other sources or aids than the declared and cited ones have been used.

Hamburg, August 2009
Patrick Un

Contents

List of Tables

Listings

List of Figures

1 Introduction and Background

Service oriented computing is an implementation agnostic paradigm that can be realized by suitable technology platform, facilitated by distributed computing architecture and compartmentalization of complex process oriented business logic into manageable, decoupled and freely composable units of application logic that are exposed as services. To retain independence, services encapsulate logic within a distinct context. This context can be specific to a business task or a business entity underlying business processes. The size and scope of the processing logic represented by services can vary.

Service oriented architecture (SOA) is an integration architecture based on the notion of service oriented computing, especially for the enterprise application domain [Erl, 2005]. Integration means first of all bringing together applications by providing connection, enabling utilization of resources, harmonizing heterogeneous systems by adapting differences and resolving conflicts, standardizing incompatible architectures, mismatched formats and proprietary platforms towards interoperable components so that one can facilitate an even more efficient use of current and legacy systems in the enterprise application landscape. Business automatic are typically realized out of business processes which are comprised of business logic that dictates the actions performed by process components. The underlying business logic can be decomposed into a series of steps that execute in predefined sequence according to specific business rules and constraints.

1.1 SOA4All Project Background

The Service Oriented Architecture for All (SOA4All[1]) is an integration project of the European Seventh Framework Programme[2] [Domingue et al., 2008] aiming to facilitate a wide spectrum of users the easier comsumption and adoption of large number of services by providing comprehensive service delivery framework to enable context based, automated service delivery. Using the principles of the service oriented paradigm for application development, adopting context management for personal preferences, managing system constraints and organizational policies, harnessing semantic web technologies and web 2.0 to automate service discovery, mediation and composition helps to incorporate the vision of semantic web into service oriented computing. The goal of this project is to advance current SOA in these aspects:

- enabling easier service consumption and turning the method of consumption more user friendly,
- using open standard to realize better interoperability,
- making service integration more customizable to incorporate user preferences and context information,
- enhancing scalability of the architecture by field test integration of million of services.

Existing enterprise SOA platforms such as the SAP NetWeaver application server platform are usually designed for software developers and experts in business process management [van der Aalst et al., 2003, van der Aalst et al., 2005] rather than regular end users. As a result, they offer rich functionalities development and detailed modeling of business processes to a rather selected group of domain experts and process

[1] SOA4All http://www.soa4all.eu/
[2] Seventh Framework Programme http://cordis.europa.eu/fp7/

modelers. Additionally, their focus is on planned, well-structured, adequately syntactically formalized and highly repetitive business processes which are often located within a single department or organization of an enterprise or among a few organizations with well established business relations. Consequently, current SOA platforms with heavyweight process oriented services usually cannot be directly exposed to general end users because they lack the necessary technological skills or business process background knowledge.

This is where SOA4All strives to achieve its goal. It aims to empower general end users easier access to service consumption. It envisages an open, dynamic service environment where a large number of actors, who might be system users, process integrators and even business process experts , can expose, combine and consume a very large number of services. To enable such a challenging vision and make the corresponding business scenarios manageable, SOA4All builds on four complimentary and proven paradigms [Domingue et al., 2008] to achieve its goal: supporting open web standards and system architectural principles for the problem of service integration on a world wide scale; using proven technologies of existing web 2.0 technology infrastructure to enable better participation, personalization and sharing of information as a means to structure human-machine cooperation in an efficient and cost effective manner; using semantic web technologies as a means to abstract from syntax driven to semantics driven, meaningful and precise service discovery and process handling; using context management as a means to capture and process user needs and preferences in a machine interpretable way so that services can be customized for specific requirements.

1.2 Motivation and Thesis Statement

The focus on use cases and business scenarios requires conceptualization and implementation of an end user service delivery platform that can deliver general web services as well as integrate process oriented enterprise services. Consequently one of the research questions of SOA4All is the investigation of how to integrate and extend process oriented enterprise services into an open, dynamic, lightweight and end user driven service platform.

1.2.1 Motivation

Enterprise services are defined as the services that are provided by enterprise computing environments that expose functionality of enterprise application systems [Weske, 2007]. Currently these services are implemented with web service technologies [Alonso et al., 2003] to a large extent. This is not necessarily the case because the functionality of the application system is provided through services which can be realized by other application technologies that conform to the service oriented paradigm. In the SAP applications enterprise services are implemented with web services technologies which expose functionality of the SAP enterprise resource planning ERP backend systems to application users.

Common perception of enterprise services is that they comprise a plethora of powerful functionalities that unleash the business functionality of the backend systems on one side, with the drawback of complex interface definitions, abundant operations and elaborate data types which must be handled accordingly on the other side. In order to utilize and integrate them efficiently, complexity in their usage must be resolved and their semantics must be understood thoroughly. Consequently ERP system based enterprise services are often referred to as *heavyweight processes* where the coupled-ness with the backend system and its business logic is high. This situation reduces the possibility of incorporating flexibility and intelligence into existing systems due to the difficulties encountered when often only process experts are able to understand and integrate these processes as mentioned previously. In short combination and service delivery are hampered by

inflexibility.

From the perspective of formal knowledge representation [Fagin et al., 2003, Levesque & Brachman, 2004] and artificial intelligence, building a more *intelligent* ERP system can be viewed as one possible solution to enhance accessibility and flexibility of ERP system and enterprise services. Such an approach means that the *semantics* of business processes and operations must be better annotated, documented and processed, for instance, using metadata to describe the semantics and constraints of business processes and employing logical reasoning on the metadata to infer implicit knowledge from existing explicit one or uncover hidden relations such as evocability and constraints of enterprise services or possible execution ordering, etc.

Another solution approach is to study the existing enterprise services in order to understand their semantics in terms of capability and invocation constraints. The motivation lies in the synthesis of dedicated process functionality satisfying specific scenarios. This can be understood as intelligent combination of existing partial functionality by using metadata to semantically annotate the existing enterprise services to enable exposing their capability more precisely so that intelligent agents can be used to discover, match and select more appropriate enterprise services to satisfy process requirements. In terms of efficiency and complexity, due to the possibility of using appropriate heuristics, matching service capability query with existing service capability descriptions regarding enterprise services can benefit from reduction of the size of search space, thus transcending the mere limitation of contemporary keyword based search service discovery and selection approach.

1.2.2 Objective

Obviously it is not practical to redesign an existing ERP system from scratch so that intelligence can be built into the system at design time. Nevertheless we are motivated to investigate the integration of enterprise services in the context of SOA4All. We approach the integration problem based on a field test business process to illustrate an integration scenario and to evaluate functionality synthesis in processes by studying the problem of automatic service composition. By inquiring research efforts in academia from the domain of knowledge representation, agent theory and technologies, artificial intelligence (AI) and logical reasoning we strive to:

A) give reasonable characterization of existing process oriented enterprise services,

B) survey existing industry solutions and academic approaches in service modeling, service matching and service composition critically and give source references to help to understand the essential and fundamental issues involved,

C) lay a foundation for complete, decidable and tractable logical account for service matching and composition and use the situation calculus formalism [Reiter, 2001a] and agent theory to tackle composition problem in a guiding business process scenario from an action theoretical perspective of knowledge representation [Fagin et al., 2003],

D) draw conclusion and give dicussion from a sound and complete composition approach to serve as an example for future attempts in enterprise service integration for SOA4All.

The fundamental research question is how to correctly and effectively compose enterprise services semantically to achieve business specific goals within a business process scenario. By semantic we mean that the properties, constraints and conditions of enterprise services are very often implicit and therefore not easy to understand. We want to inquire fundamental issues of service matching and composition and from such a perspective evaluate integration of existing enterprise service. We address and investigate the fundamental issues of the following:

I) What are enterprise services and how can they be composed and coordinated correctly?

II) How can state constraints in terms of preconditions, invariance and postconditions of enterprise services be correctly and completely characterized?

III) How can we characterize a set of enterprise services in a composition such that we can get a correct sequence of executions of these services?

IV) What theoretical framework is expressive and powerful enough to characterized properties, constraints and conditions of enterprise services?

V) Assume we possess an appropriate framework and that we observe operations of enterprise services as executable actions, what kind of action theory is needed and how can we axiomatize a domain?

VI) How will such an axiomatization affect enterprise service composition and when do we know that we have obtained a correct composition?

If we assume a theoretical approach for composition of enterprise services to integrate them into a business scenario, at first it seems that it is not on par with the lightweight thought of service delivery in SOA4All. Providing sound, complete and correct approach to tackle existing integration problem has often been falsely accused of tending to be heavyweight or arcane unfortunately because understanding such theoretical approach would assume existing knowledge in relevant academic discipline. However on the other hand we understand that since the lightweight thought of service delivery in SOA4All does not actually prohibit more formal approach nor is it contradictory to the attempt to correctly delimit the problem domain in order to investigate essential and fundamental issues underlying the problem. Using correct and proven theoretical frameworks of knowledge representation formalism and logical reasoning has convinced us to believe that the intrinsic semantics of enterprise service often incur implicit interaction constraints and condition for execution which can only be correctly captured by an appropriately sufficient logical formalism. Domain dynamics and states must be axiomatized using an appropriate formalism that is invented for the dedicated purpose of describing the dynamics within involved interacting systems. It is like the metaphor of using the right tools to do the right job. We are convinced that based on an appropriate axiomatization of the problem and using logical reasoning technique we can derive the correct results. We want to show that business process experts should also share this view in future SOA4All integration scenarios.

1.2.3 Contributions

We hope that future research in the area of process and service integration can be stimulated to considering the problem from a new perpective by grasping the essential and necessary substance of the issues rather than solely relying on conventional software technologies. We strive to bring up attention to some appropriate formalisms to tackle the current problem and argue that more formal approach ín problem solving must be considered without bias.

Our approach is understood as a consolidated impulse for SOA4All to inspire contemplation on the essential issues of enterprsie integration. It is novel in the sense that the current integration project goal prescribes heavyweight enterprise services that are complex and inflexible; therefore as such they require correct and complete account of their process semantics which are not representable in mere lightweight syntactic based descriptions. Very few past works exist on marrying the notion of using decidable action theory of the situation calculus to model the intrinsic dynamical aspects of enterprise services with regard of the practical aspects of process-orientation. None exists that axiomatizes the service domain and simultaneously observes aspects of intrinsic service constraints within an existing non-semantical process oriented ERP setup.

1.3 Thesis Structure

This thesis is structured as follows: after the background information is introduced in chapter 1, we give a comprehensive account in chapter 2 on existing SAP enterprise services, their information and behavioral model as well as a description of the runtime system which ensures that process and service actions trigger the correct state changes. In chapter 3 we review and survey different approaches for semantic service matching and automated service composition in the industry and academia; and subsequently giving a brief critical comparison of these approaches. In chapter 4 we inquire the formal foundations of service matching and select two exemplary formal approaches to describe enterprise service composition and integration. In chapter 5 we introduce the guiding process scenario in the SOA4All context to illustrate the appropriateness of the introduced formal action theoretical approach in solving the composition and integration problem correctly and efficiently. Finally in chapter 6 we give dicussion on the lesson learned and outlook future works.

2 Enterprise Services

In this chapter we relate the concept of business process management and enterprise resource planning systems to characterize SAP enterprise services and shed light on some issues of service integration. We begin with an account on enterprise information system in general and continue with brief introduction of SAP enterprise services, its underlying information model, process and behavior model.

2.1 SOA Enterprise Service Characteristics

2.1.1 Service Oriented Computing

Service oriented architecture (SOA) is one of the principles that is adopted in SOA4All [Domingue et al., 2008]. Service orientation architecture is not a software library nor merely a framework but a distinct implementation-agnostic principle to design application in a service centric way. It represents a form of technology architecture that adheres to these aspects:

- loose coupling: services maintain only necessary relationship that minimizes dependencies that are required to retain awareness among one another, i.e., services must be designed to interact without unnecessary tight, cross-service dependencies;
- service contract: services adhere to an agreed upon contract of communication in order to advertisement valid input, output data as well as expose their operations in a standardized manner;
- service autonomy: services are endowed control over the business or process logic that they encapsulate;
- coherency and abstraction: services are described in an agreed upon service contract without exposing other necessary or irrelevant implementation details;
- reusability: modularization of processing logic contributes to better functionality management with the intention to promote better reuse of generic modules;
- composability: a set of related services can be coordinated and assembled to form composite service which can satisfy more complex functional requirement. Composability allows business logic to be encapsulated and representable at different levels of granularity;
- statelessness and statefulness: stateless services are intended to minimize retaining information for a prolonged period belonging to specific service activities or sessions in order to simplify service implementation. On the other hand stateful services can be involved in more sophisticated interaction patterns by appropriate management of stateful conversational state;
- discoverability: well annotated services are designed to facilitate explicit description of either syntactic or semantic based information so that they can be searched for and selected efficiently via available discovery mechanisms.

When realized through the current web services technology, SOA establishes support of open, standardized and interoperable communication of business process oriented services and contributes to enhanced automation domains of an enterprise [Erl, 2007, Erl, 2009]. Although there is often the misconception that when an application uses web service technology, it is service oriented, in fact service orientation is not solely realizable using web services. Instead web services have been used as a vehicle to facilitate implementing a service architecture that is currently at the heart of SOA implementation. It does not need to be web services at all because service oriented applications and platforms may be implemented using other novel, even more suitable technologies in the future rather than client-server based distributed systems.

Contemporary SOA represents an open, extensible, federated and composable architecture comsisting of partially autonomous, interoperable, discoverable and to a large extent reusable services that are implemented using web services technology. Service oriented architecture can be conceived as a high level abstraction of business logic and process technology which when implemented appropriately results in a possible loose coupling between application domains. Contemporary SOA is also an evoluation of existing distributed computing platforms [Lynch, 1997].

The collective business logic that defines and drives the applications of an enterprise is an ever evolving entity that is constantly undergoing changes in response to external and internal change of requirement. The dichotomy of this evolving entity is the division between *business logic* and *application logic*. Business logic is a documented implementation of the business requirements that correspond to the business area of an enterprise that is reflected in the close alignment of business logic with domain specific models. Business logic is generally structured into processes that express these requirements along with a set of constraints and dependencies. Application logic is an automated implementation of business logic that are organized into different solutions based on their underlying technology. Application logic expresses workflows of a business process through specification of *dataflow* and *control flow* [van der Aalst et al., 2005], i.e., dataflow specifies relevant data pathes which essential information, either input or output flows through the process during the execution of the application logic while control flow clarifies the move of the thread of control of a process during execution.

2.1.2 Business Process Fundamentals and Enterprise Services

The service oriented paradigm applies to enterprise logic by introduction the concept of encapsulated services to properly expose the functionality of business processes of an enterprise information system. A *business process* consists of a set of *activities* that are performed in coordination in an organizational and technical environment. These activities jointly realize a *business goal*. Each business process can be instantiated on its own or interact with other business processes at different organizations [Weske, 2007]. *Business process management* is a discipline of study that investigates concepts, methods and techniques to support the design, implementation, administration, configuration, enactment and analysis of business processes [van der Aalst et al., 2003]. The basis of business process management lies in the formal and explicit representation of business processes, their corresponding activities and execution constraints such as execution order or necessary conditions leading to state changes, etc. For managing business processes, a dedicated *business process management system* can be used which is driven by the explicit process representations to coordinate the enactment of business processes. For the purpose of modeling business activities, business process models are used. Such a model consists of a set of activity models and execution constraints between them. A *business process instance* represents an instantiated view a predefined business process in its operational manifestation which contains activities instances. The relation between a business process model and process instances resembles the relation of a blueprint and concrete edifice of business processes built from the blueprint.

Tradition enterprise resource planning systems are designed to span large parts of the business processes of an enterprise with consistent, centralized data storage. They store data in integrated databases and provide business functionality to application clients via exposure of these functionality over an application server to provide access to clients. Enterprise systems architecture is the underlying software architecture that supports this type of application setup and it is mainly based on process oriented workflow information systems which can be extended with support of service orientation by providing the functionality as enterprise services. They capture functionality with a business value that is implemented by software service instances a provider platform and is ready to be used. The service oriented paradigm is the main influence factor for enterprise information system currently and enterprise services architecture is based on the understanding that complex applications are increasingly built on top of existing business processes and functionality that are characterized by the added value of enterprise services through standardized interfaces. Driving forces for development of enterprise services are mainly change that dictate business scenarios and the state of a value chain; consequently there is a need to increase enterprise system transparency and computer mediated interaction with customers and corporate suppliers with the increased perception of the the presence of a corporation through the services it provides. These services exposed to customers can be realized in the form of enterprise services that provide access to functionality of the backend application systems.

Advance in technology has paved the way for enterprise services. The major milestone of this development is the commercialization of full suites of enterprise application products with built-in enterprise services which decompose processes and workflows functionally as coarse-grained services that are provided in customizable middleware that are available nowadays, e.g. ERP application suites from vendors such as Oracle and the SAP Netweaver Suites products.

2.2 SAP Enterprise Service Information Model

Existing SAP enterprise services are developed in the way that inherits the general notion of enterprise services described in the previous section. The technological infrastructure for implementing the SAP enterprise services is the web services technology platform using standardized interface description such as WSDL, transport protocol over HTTP, synchronous or asynchronous invocation pattern and registry technology such as UDDI. Currently the WS-$*$ stack of extended web services specifications and standards such as WS-BusinessActivity, WS-CDL, WS-Coordination, etc. are not yet completely supported by the enterprise service middleware, though limited support of WS-BPEL and a set of other important web services transaction standards have been built into newer releases of the NetWeaver Suites products to support better realization of service oriented architecture. What characterizes these SAP enterprise services is the fact that they are organized into bundles according to the business domain and area, such as banking, CRM and SCM, etc. for which these enterprise services are designed. Understanding the intrinsic *information model* of the underlying SAP enterprise resource planning systems is crucial in order to get a notion of how data is structured within business processes and how functionalities are implemented for these enterprise services.

2.2.1 Business Objects and Process Components

The aim of enterprise modeling is to clarify business process workflows and dataflows by using a systematic approach to structure, organize, visualize and document the relationship, dependencies and interactions between components of an enterprise system. In order to facilitate reuse and make the structure of enterprise data more transparent and manageable, the fundamental idea is to use a clear

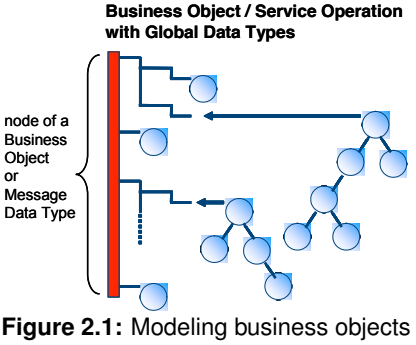

Figure 2.1: Modeling business objects

abstraction to model data in detail that is independent from the implementation. The data of each process workflow is modeled using *business objects* in SAP systems [Snabe et al., 2009]. A set of business objects are shown in figure 2.1. Business objects describe the data of a process as fundamental idea of distinction between the data and business logic implementation and are structured to enhance good modeling practice of the business process, avoiding duplication of modeling, overlapping definitions of data and redundant information storage. Business objects represent specific views of data of a well-defined and outlined business area and are the central point of the modeling principles used by SAP systems. Business objects are the fundamental modeling unit that can be combined to form more complex data structures. The duration of a business object is another aspect to characterize in data modeling with the difference between transaction data and master data:

- business objects that work with transient data during the course of a business process represent transaction data. They are called *business process objects*;
- business objects that work with essential persistent data within the entire system over a long period of time are called *master data business objects*.

Enterprise services always need to work with and access data. Therefore the modeling of enterprise services are intrinsically linked with the modeling of business objects, both business process objects and master data business objects. The business process modeling with business objects is carried out on the ARIS modeling platform products[1]. In an ARIS model, an abstraction using *process components* is used to work with business process modeling to identify self-contained part of a business value chain which is relevant and crucial in subsequent implementation of cross organizational business-to-business processes [Snabe et al., 2009]. A process component in SAP system is defined as the part of a value chain that is performed for a specific business area in an enterprise and it acts as a unit to group business objects with each business object belongs to one process component. Additionally the level of granularity of modeling using process components is also relevant because the modeling of business objects is closely connected with the granularity chosen which is important to ensure that there is no overlapped part of process components containing duplicated business objects that are accessed in different process components.

Accessing data in a process component means access to the relevant business objects of the component. There are two types of access patterns: asynchronous access to process component is used for inter-process communications between process components of different business areas in different value chains; synchronous access depends on other process components of the same business area within an application. Access to process components is modeled in ARIS as service *interfaces* and *operations* that access business objects with each operation is assigned to one business object. A service interface is used to group operations in order to expose business object data access operations to external systems.

While the main purpose of a process component model is to describe the inner workings of a business process component specifying the relevant business objects to represent data as well as the corresponding service interfaces and operations to access them, a *deployment unit* is used to group all process components that belong to a specific business scenario so that they can be installed together and configured for deployment. In the *process component integration model* deployment units are modeled in relation with the corresponding process components on the ARIS platform.

2.2.2 Core and Global Data Types

Business objects represent a kind of semantic structure to model business process with specification of data access operations defined in service interfaces, on the other hand a conceptual data model has been intro-

[1] ARIS Platform http://www.ids-scheer.de/de/ARIS

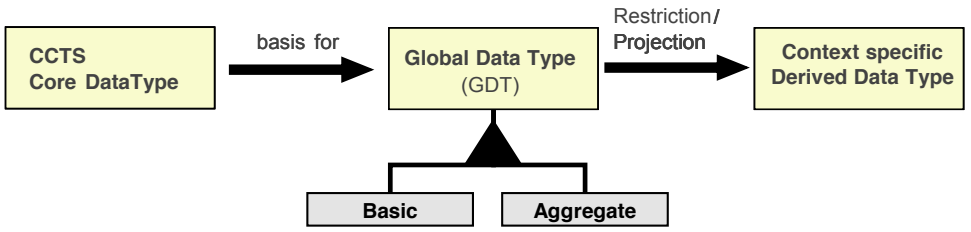

Figure 2.2: Relation of data types in the information model

duced to structure data types within SAP applications. *Core data types* are international standardized data types which are modeled according to UN/CEFACT Core Component Technical Specification (CCTS/ISO 15000-5). They do not contain business semantics and define exactly one primary component known as *content component* across all types of SAP applications.

Core data types are syntax neutral and represent an atomic and most generic pieces of information in a business process model. Core data types are based on primitive data types such as primitive types of the W3C XMLSchema and yet are different than general primitive types because they carry attributes and properties that further define a concrete value domain. A data type repository is used to store the representations of core data types which are defined using verbal representation terms. Each core data type can be identified with the representation term which is not specific to SAP application only because it is detached from specific business semantics details.

Global data types are divided into basic and aggregated types. While the basic type is built directly on core data types, the aggregated data type contains list of elements of global data types, i.e., it can be seen as a composite type for the global data type. Global data types contain business semantics that is used cross application. Figure 2.2 shows the relation between the mentioned core data types and global data types. Basic business semantic is built on top of global data types that has context-specific business process. These context-specific data types are derived from global data types.

Common business semantics of global data types are modeled in a *template* in ARIS model which specifies all the elements and attributes of a global data type without redundancy where attributes define elementary features of the underlying data types. Templates have no implementation and it ensures the consistency of semantics of global data types to which they describe. Global data types specify value range which constraints the value of attributes and content they represent. Based on the fact that global data types are built from core data types, they inherit the data structure and integrity conditions from their constituents.

Dataflow for enterprsie services is based on the notions of business objects, core and global data types as well as the corresponding operations semantics. Figure 2.3 shows in a comprehensive way the semantic structure of the business object view of enterprise services, illustrated with the example business object representing purchase order. Underneath the semantic structure layer, corresponding entity specific and global data type are shown in relation to the semantic structure of a process component for the business logic purchase order processing.

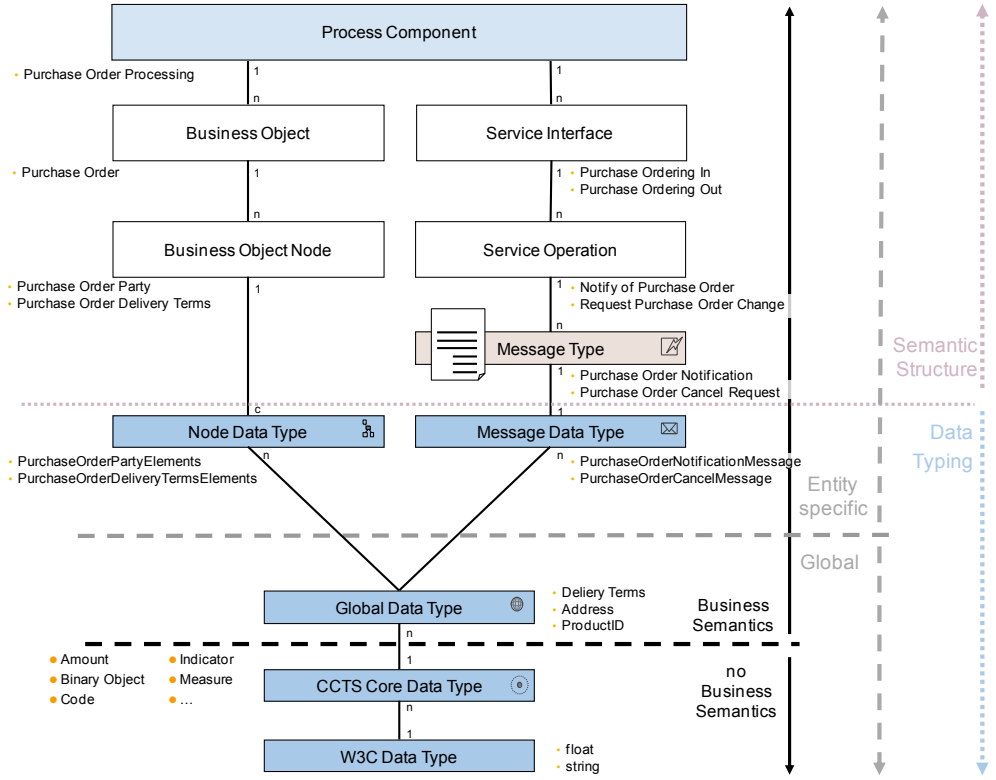

Figure 2.3: Semantic structure and data types of business processes

2.3 SAP Enterprise Service Behavior Model

We turn our attention to description of service behavior. Although the previously described information model of SAP enterprise services can be characterized with an existing upper service ontology that describes each data types semantically, such ontology is not sufficient when it comes to describe the service semantics in terms of behavior. Beside dataflow and control flow which characterize enterprise services, if we adopt an abstract perspective and view such services as black boxes which require certain constraints to hold and some preconditions satisfied before they can be executed, and if we also view the result of the process steps during execution, the constraints for executing the steps are essential and must also hold; the same is true for effects which service execution and process steps have on the states of the overall application domain and on the state of the data in the backend. It is clear that we must both state the constraints, preconditions, invariance and effects of enterprise service with clear specification, so that they can be combined and coordinated in a way that is coherent with their execution semantics. As we know that enterprise services expose functionality of the underlying enterprise application systems, therefore its behavioral semantic is also intrinsically related to that of the business processes.

2.3.1 Service Constraints

Operations on business objects change the state of the business objects. The state is called in a SAP application a *status* and it is a business object attribute and a modeled entity that represent the life cycle of a business object or the result of processing steps within a process. It represents additionally the preconditions and effects of processing steps. State of a business object can be viewed as a kind of control to facilitate decisions at different process steps within processes and therefore it influences the behavior of processes as

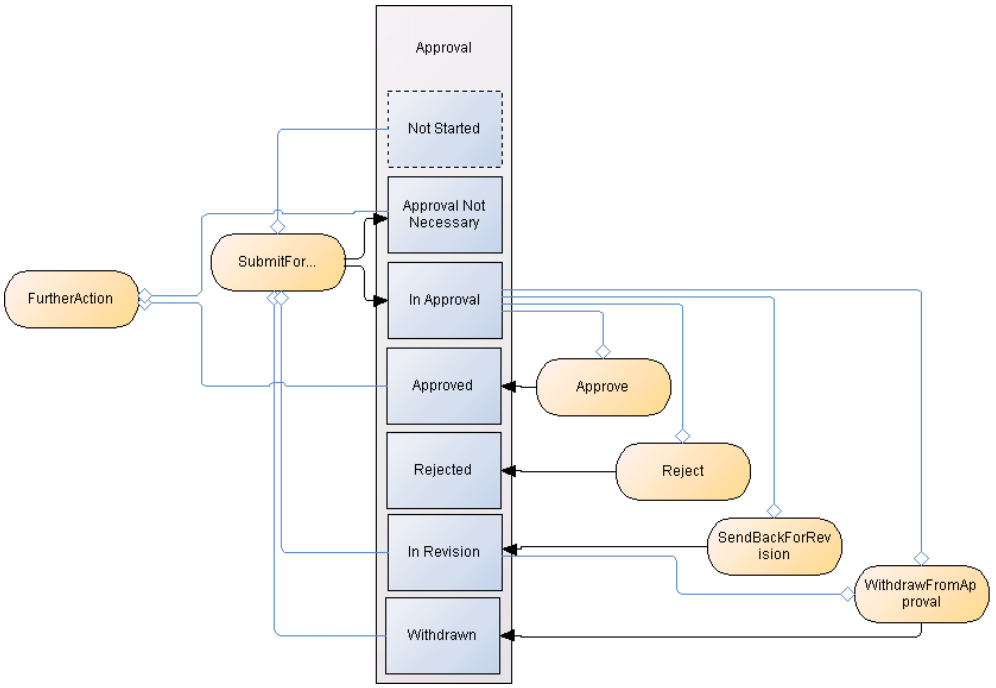

Figure 2.4: Status and actions of a business object

a whole.

The state of business objects can be change by an *action* which abstractly represents any system internal or external activity that has an influence on the state and consistency of business objects. An example of an business object called *Approval* with the actions shown in the round boxes which can change the set of status within the business object are illustrated in figure 2.4. Notice that actions which point out as arrow that set a status while diamond head is an enabler for an action to execute. An action requires an implementation that performs the corresponding business logic associated with the action. This action implementation is responsible of transforming the attributes of the business objects which correspond to the status and subsequently performing business logic to determine possible resulting status. Application constraints therefore describe which actions are allowed to be performed when specific status are reached during certain process steps.

2.3.2 Status and Actions Model

In SAP enterprise services, the status and actions management runtime is a control and monitoring system which guard constraints and ensure that the life cycle of business objects in a process is correctly observed and that operations on the business object will be conformed to the predefined constraints. Constraints in business process are often caused by real world events that are part of the business process, for instance user input or certain activities that are irreversible. Therefore safeguarding the consistent and correct status transition is crucial in the runtime system. Constraints on business object, and process component represent the core process entities by defining constraints between status changes and corresponding actions.

For modeling processes that underly enterprise services, one can proceed with identification of process steps of a business process, review the steps and identify relevant actions and status values. For each process step, one models status, actions and constraints diagrammatically in a graph structure as shown in figure 2.4;

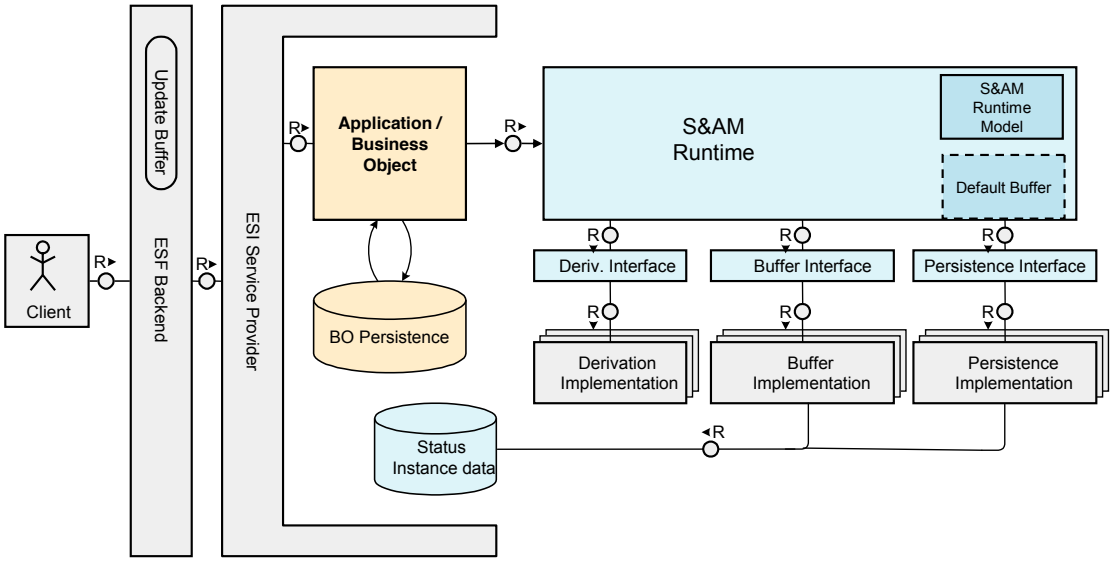

Figure 2.5: Status and actions management runtime system architecture

subsequently all identified process steps are assembled so that the dependency constraints, preconditions and postconditions among the process steps are modeled appropriately.

Clearly an SAP enterprise application must access the status and actions management data during runtime. Therefore it is necessary to persist this data. This is where a status and actions schema comes in. A status schema consists of status variables, a set of allowed status transitions, the set of preconditions that are necessary to influence status changes and allowed actions of business objects. It is used to group status, constraints and actions to the corresponding business object. At design time a status schema is provided with all known status while during runtime only certain status values will appear as current status values of a business object within a process instance. During runtime a module called the status and actions management runtime is responsible for intercepting the operations on business objects of a process instance so that constraints of each process steps can be checked and enforced and executed operations are guarded for data consistency and integrity.

Figure 2.5 shows the runtime overview of a status and actions management runtime module in action. The figure also indicate the relation between the runtime module with enterprise service instance (ESI) runtime backend which is responsible for creating, configuring and running enterprise services. The ESI runtime and service framework of an SAP enterprise application is shown on the left hand side together with a depiction of client service call. The status and actions management runtime accesses status schema and relevant persistent information from a backend status instance repository in order to compute correct status values and corresponding status transitions for the different operations on the business object which themselves are exposed to the service clients as enterprise services.

With this brief explanation on the characteristics of enterprise services and their underlying information, process and behavioral models, one understands that in order to effectively utilize existing enterprise services with the goal of providing solution to complex business process oriented problems, one must understand the behavioral semantics of these services. Furthermore in order to generate more customizable and complex functionality from existing, self-contained and often domain-specific enterprise services, it is necessary to find effective ways to combine the individual services together in a sensible manner so that constraints are

observed and correctness is guaranteed. This is where the problem of composition enterprise services comes in. We will see some of the existing approaches of semantic service composition synthesis in the next chapter.

3 Survey of Related Works on Service Integration

In this chapter we review and analyze a selection of existing research approaches and efforts as well as relevant publications in the literature in the domain of *service modeling* and *service composition* which we collectively summarize and regard as necessary prerequisite steps towards achieving well-defined *service integration*. We do not intend to provide an exhaustive account of all available approaches in the literature but a subset of those which are relevant to our research question. The survey in this chapter is mainly divided two dimensions: critical explanatory reviews on service description and modeling approaches[1] and survey on existing approaches of service composition synthesis both across industry and in academia circles. The last section of this chapter provides a comparative juxtaposition of these solutions and approaches according to a set of chosen criteria to highlight commonalities and differences among them in order to identify issues about what has been researched and what are the strength and potential weakness of these approaches.

3.1 Service Modeling

Conventional web services and enterprise services are common in descriptive aspects in a variety of ways. Conceptually they are programs that export their description model which is not defined with a strict formal approach generally. These solutions are mainly found in the industry. In the academic research, there have been efforts to combine semantic web technologies with web services to provide metadata based service annotations to conventional web services.

3.1.1 Service Characterization Aspects

When modeling service beyond a pure syntactic dimension, it is necessary to identify some other dimensions for classification of service description. We distinguish the following aspects:

1. *service interaction model*: it represents the interaction form with a service client. We observe three types of interaction patterns:
   - *monolithic interaction* characterizes service operations that are described in terms of legal and executable operations only without exerting restriction on the order of execution of these operations;
   - *sequential interaction* characterizes a temporal sequential model of description of legal operation executions where temporal order of the executions are observed linearly;
   - *tree-based interaction* characterizes a branching model of temporal execution of service operations based on the *choices* a service presents to execute in the next step after each execution of a operation.
2. *controllability*: it refers to the possibility of fully controlling a service based on the operations it executes in terms of knowing the available choices of executable operation at a certain execution step

[1] This also includes service matching based on a certain descriptive model of services

after a previous operations has been executed. Another term of full controllability is *deterministic* service. In contrary if given a certain state during execution, the choice of executable operation for the next step is sometimes not determined due to for instance abnormality of operation behavior or irreversible failure such that the service cannot be sure to offer certain choices. In such cases, we called the service to be only partially controllable or a non-deterministic service;

3. *state observability*: it characterizes the transparency of internal states of a service in terms of observable states available to external services or other enterprise components;
4. *dataflow awareness*: it characterizes the property of a service whether it deals with dataflow and other data in form of input and output parameters.

3.1.2 Service Description and Modeling

Traditional proposals for service modeling and description from the industry are these following languages:

Web Service Description Language (WSDL) is a conventional syntactic approach for describing web services [Chinnici et al., 2007a, Chinnici et al., 2007b] which mainly describes services from syntactical data type centric aspects with an emphasis on definition of operation signatures and exchanged message formats. It is a static representation of a service because it does not describe behavior of a service. Incoming and outgoing messages are declared using data types based on the XML schema language. It provides syntactic mechanism to locate service through URI, specify concrete transport protocol to exchange messages and concrete mapping between abstract method definition and concrete protocol and data format.

Web Service Conversation Language (WSCL) is a proposed language [Banerji et al., 2002] to specify and support conversational state of web services based on XML syntax [Bray et al., 2008]. A WSCL document views communication as a web service conversation and specifies the exchange order of legal instances of XML messages in a conversation.

Web Service Choreography Interface (WSCI) is a proposed language [Arkin et al., 2002] which describes the observable coordination behavior of web services from client's perspective with the purpose to represent temporal and logical dependencies of exchanged conversational messages using XML syntax.

It can be observed that WSDL provides a static representation of implementation-independent interfaces of web services and has become widely adopted. It encodes a monolithic interaction model without restriction on the order of execution of the operations specified in the WSDL interface. In comparison WSCL and WSCI provide support of tree-based interaction and explicitly deal with data and fully controllability of the services they describe.

The **Web Service Choreography Description Language (WS-CDL)** is a process oriented language [Austin et al., 2004, Kavantzas et al., 2005, Ross-Talbot & Fletcher, 2006] which has been proposed to express collaboration between participant services with appropriate specification of their conversations. It is a language which is based on π-calculus[2] with the behavior of each participant services in the collaboration being described through a sequential finite state transition system, interacting with the others and sharing resources through predefined channels. WS-CDL provides a formal model [Burdett & Kavantzas, 2004] with constructs for service communication, decision choice support, concurrency and iteration which have a precise formal semantics based on corresponding π-calculus constructs. Formal verification of livelock and deadlock is also supported.

[2] π-calculus is also known as process calculus or process algebra which is originally proposed by Milner [Milner, 1999].

Likewise in the business process management domain, there have been efforts to develop other languages to incorporate support of business process artifacts into service modeling, such as XLANG, WSFL, XPDL, WS-BPEL [Barreto et al., 2007, Alves et al., 2007]. While some of which have been adopted for process orient service model, they suffer from lacking of well-defined semantics [Grüninger et al., 2008] which provide unique and unambiguous interpretation of the specifications though the Business Process Execution Language (BPEL) family of languages have been widely adopted.

Comparing to the mostly syntactic description solutions in the industry, many semantically well-defined approaches have been proposed in academic research circles. The **First-Order Logic Ontology for Web Services (FLOWS)** [Grüninger et al., 2008] and it currently has been submitted to W3C known as **Semantic Web Service Ontology (SWSO)** [Battle et al., 2005b, Battle et al., 2005c] and related **Semantic Web Service Language (SWSL)** [Battle et al., 2005a]. It is a framework that is comprehensive with respect to web service features towards automated service discovery, verification or composition by providing a well-defined first order logic based semantic description for services. It possesses a sequential interaction model which is geared towards specification of temporal order of the executions of services.

One of the prominent earlier efforts in the direction of semantic service description and modeling with similar goal is the **OWL-S** framework for semantic markup and annotation for web services [Martin et al., 2004a, Martin et al., 2007, Martin et al., 2004b].
OWL-S is based on the W3C **Web Ontology Language (OWL)** recommendation [Bechhofer et al., 2004, McGuinness & van Harmelen, 2004, Patel-Schneider et al., 2004, Smith et al., 2004] and is developed as a dedicated service ontology based on the expressive semantics of the underlying language. Based on the core semantic web technology [Cardoso, 2007, Kashyap et al., 2008], OWL is a sufficiently expressive behavioral model to represent the static aspects of service description as well as a black box view of service precondition, invariance and postconditions of operations towards the state of services and relevant features perceivable in the external environment. OWL-S is a typical dedicated ontology for semantic web services that is built around an *upper service ontology* consisting of the following main concepts:

- a *service profile* in OWL-S is a description of service capabilities that use the constructs of the OWL language to express service inputs, outputs, preconditions and effects. The service profile is geared towards semantic reasoning support for precise and effective service discovery;
- an OWL-S *service model* specifies the service process model, i.e., as similar to many process oriented services[3] the underlying business process behavior directly influence the behavior and semantics of the enterprise services in terms of controllability, dataflow, control flow and constraints of execution of the intrinsic service operations. OWL-S service model is aimed towards enabling automated service composition and execution;
- an OWL-S *service grounding* specifies how the semantic web services can be accessed with description of the communication protocol, data marshaling and serialization and it relates semantically described service message types to grounded WSDL messages, i.e., the messages on the syntactic level.

Another effort in semantic service description and modeling is **Web Service Modeling Ontology (WSMO)** [de Bruijn et al., 2005a, Arroyo et al., 2005, de Bruijn et al., 2005c, Lausen et al., 2006, Studer et al., 2007] and the related **Web Service Modeling Language (WSML)** [de Bruijn et al., 2005b]. The proposed ontology and language can be used to specify abstract conceptual of service descriptions with focus on interoperability of semantic web services. This proposal has provided a reference implementation of a *semantic execution environment* (SEE) runtime framework called **Web Service Execution Environment (WSMX)** [Bussler et al., 2005], to support deployment of semantic web services to showcase WSMO. The service

[3]For instance, SAP enterprise services are examples of typical process oriented services.

ontology WSMO is built on the basis of frame logic [Kifer et al., 1995] which allows a WSMO ontology to use attributes to describe service relevant concepts in details. WSMO is based on a combination of semantics of description logics (DL) [Baader et al., 2007] and Horn logic as well as logic programming [Perrin et al., 1990] with a reduced expressive variant based on description logic programs (DLP) [Grosof et al., 2003]. Generally WSMO has four main components:

- a *goal* is a formulated objective or client's desire for certain functionality w.r.t. a service. A goal consists of a specification of the state transitions based on Horn logic rules which represent the preconditions regarding the state of a described semantic service before it is executed, the states that are kept invariant, the postconditions that must hold, as well as the effects after the service is executed. It is an abstract state machine approach for specification of service behavior that is inline with WSMO;

- a set of ontologies which formally describe all the components, parameters, etc. of the described semantic web service. It is a well-defined mechanism to semantically annotate services so that every constituents of the services can be enriched with semantic metadata for interpretation. In contrary to the OWL-S approach to adopt OWL based ontology expressions, WSMO has adopted an object-based frame logic approach to specify service ontologies;

- a set of *mediators* [Lausen et al., 2006] which is conceived as dedicated first class components in WSMO based semantic web services to resolve incompatibilities between service ontologies, service interfaces, service goals, etc.; WSMO mediators do not dictate concrete implementation details about how these mediators must be built, rather they stay on an abstract level and specify what elements must be available in the mediators and what such mediators achieve to perform functionally. Mediators can be implemented as software components that are deployed in the WSMX runtime environment;

- a set of *semantic web service descriptions* using the defined service ontologies, they specify the dedicate service ontologies for each semantic web service consisting of functional descriptions expressing service capabilities which are summarized in a service interface syntax using WSML. Moreover the orchestration and coordination details between interoperable semantic web services with each other are specified also in the service descriptions using corresponding WSML language syntax. In order to resolve incompatibilites which can arise during service interactions, the mentioned mediator specifications can also be included in each service description. WSMO service descriptions are essential to express the behavioral model of semantic web services.

The WSMO ontology is expressed using the related Web Service Modeling Language (WSML) literally [de Bruijn et al., 2005b, de Bruijn et al., 2008, Lausen et al., 2005, Steinmetz et al., 2008, Toma et al., 2008] which exemplifies and reflects the semantic layering [de Bruijn & Heymans, 2007] corresponding to the semantics and expressiveness of WSMO ontology. WSML supports a frame logic based surface syntax allowing definition of abstract concepts, abstract relations, instances of concepts, instances of relations as well as logical axioms that are based on a rules in Horn logic syntax [de Bruijn et al., 2005d, Lausen et al., 2005]. Moreover WSML also has an abstract syntax and intrinsic semantics based on Hoare logic [Hoare, 1969] by Charles Hoare and Horn logic as shown in [de Bruijn, 2008]. Serialization of WSML is supported in XML and RDF syntax [de Bruijn et al., 2008, Toma et al., 2008]. While the four former constructs focus on the modeling of the constituents such as input, output and instances of semantic services, the latter axioms supports definition of logic rules and conditions for state transitions which are essential in the interaction and behavioral model based on abstract state machines that are intrinsically defined in the WSMO service ontology.
While the DLP variant is weak in expressivity in terms of the descriptive power of service description but is decidable [Grosof et al., 2003] computationally, the more expressive variants of the WSML language such as WSML-Full or WSML-Rule based on Horn logic programming can suffer from undecidability because of unrestrained expressiveness in logical syntax. Since WSML possesses a combination of semantics in logic

programming and description logics, therefore it requires both type of reasoners to support semantic service discovery and mediation. The interaction model of WSMO/WSML semantic web services follows the monolithic model in the dedicated ontologies and the sequential interaction model in the service mediation and service descriptions based on abstract state machine state transitions.

In academia another interesting web service description modeling approach is published in the works of the group of **Ambite, Takkar et al.** [Thakkar et al., 2002, Ghandeharizadeh et al., 2003, Thakkar et al., 2003, Thakkar et al., 2004, Knoblock et al., 2005] where the interaction model is a monolithic one and services are modeled as view over existing data sources. Services are described by their input and output parameters, binding patterns and constraints on the data sources, which directly characterize the output data that the services return.

Further research approaches have characterized service modeling which exports the behavior of services which has proved to bear significant relevance to the latter described composition of services. Works such as explained by **Hull et al.** in [Bultan et al., 2003, Hull, 2005, Hull et al., 2003] and the group **Deutsch et al.** [Deutsch et al., 2009, Deutsch et al., 2007, Deutsch et al., 2004, Deutsch et al., 2006a, Deutsch et al., 2006b] as well as **De Giacomo's group** in Rome [Berardi et al., 2005b, Berardi et al., 2003a, Berardi et al., 2003c] together more or less characterize services from an observable, deterministic or non-deterministic perspective that focus on conversational descriptions. Many of these proposed approaches follow the tree-based interaction model and consist of detailed specification on observability and controllability of services.

In [Bultan et al., 2003] services are modeled as possibly non-deterministic Mealy finite state machines that are able to send or receive messages to or from service peers according to a predefined communication network configuration. Services communicate and evolve asynchronously and are equipped with a bounded queue buffering incoming messages while if the queue is zero-length, services communicate synchronously. The behavior of a service is summarized as sending messages, receiving messages, consuming queued messages or simply performing no operation. These operations are mutually exclusive and yield when performed state transitions w.r.t. the services.

A similar framework which incorporates both message exchange among services with exported behavior and interaction with a shared database is the COLOMBO framework [Berardi et al., 2005b] where services are represented as Mealy deterministic finite state machines and are equipped with queues and asynchronously exchange messages with other peers. The framework interacts with a database through atomic and possibly non-deterministic processes with OWL-S as description language.

From a modeling view point that resembles the previous approach, **Pistore's group** has also proposed in [Pistore et al., 2004, Pistore et al., 2005a, Pistore et al., 2005b] a behavioral description of services as non-deterministic finite-state processes which are specified in WS-BPEL [Alves et al., 2007] and abstractly represented as finite state transition systems that are partially observable which mean that states are not completely known during runtime. Each state of the transition system represents a service internal state which is characterized by the set of operations the service offers and transitions represent the state changes that a service performs when an operation is executed. The non-determinism incurred by partial observability models partial knowledge of domain in terms of uncertainty about possible outcomes of executing operations, however partial observability is a correct way to model the inability to observe all properties of the internal states of services that are ubiquitous in many services; this approach follows the tree-based interaction model.

3.1.3 Service Matching and Discovery

In industry a standard solution for service discovery has been **Universal Description, Discovery and Integration (UDDI)** which is a registry specification proposed by a consortium of several major industry vendors. The repository is referred to as UDDI registry which is administered centrally and can be physically distributed. A service provider publishes service descriptions as WSDL to a UDDI registry which store the descriptions and replicate the information accordingly. The core of a UDDI registry consists of the conceptual components of *white pages* which contain business information about the service provider, *yellow pages* which contain classifications of services in various taxonomies and *green pages* which provide technical information about the published services [Alonso et al., 2003]. A service consumer can consult the UDDI registry by searching based on keywords or service name. Found results are returned with links to invoke the services to the service consumer. The major drawback of UDDI is scalability issues and most of all the inability to precisely expose the functionalities and capabilities of services in the registry beside just classification according to certain business taxonomies or verbal descriptions.

In academic research, many research efforts have concentrated on finding appropriate ways to expose and declare the capabilities of services semantically so that automatic logical reasoning and inference mechanisms can be used to match for most appropriate services bearing the capabilities that are desired.
A standardized model of semantic matching of services has been introduced in [Paolucci et al., 2002] where an OWL-S (then DAML-S) based service profile matching algorithm is proposed to define semantic matching model in conformity to the semantic matching notion of services described in description logics. Five types of semantic matches are identified with the preference and degree of matches between capabilities formulated in a query and a set of service profiles. They are *exact match*, *plug-in match*, *subsume match*, *intersection match* and *non-match* which will be described in detail when we formalize matching in chapter 4. The semantic matching model of semantically annotated services has become widely adopted in academia and research.

In the works of [Agarwal & Studer, 2006] of **Rudi Studer's group**, it has been proposed to semantically annotate WSDL documents with rich semantic descriptions. Moreover a brief reasoning algorithm is proposed to match services and return either matching services or a set of conditions under which a services offer the desired functionality requested in the matching query. A goal based and equvalence based formalism for matching services is propose in [Agarwal, 2007] which allows to describe services with involved resources. Goal based matching deals with specifying constraints on desired web services and then finding those services that satisfy the constraints, whereas equivalence based matching supports the view that an existing service is replaced by another service without changing the overall behaviour of the system hosting the service. An expressive formalism is developed to specify desired constraints on a service with a sound and complete algorithm to check whether a service description fulfills a goal. Furthermore regarding functionalities of services algorithm that shows how two service descriptions can be checked for equivalence is developed.

In the works of [Noia et al., 2008, Noia et al., 2007], the researchers have described a formal non-monotonic logical approach of semantic matching of services and algorithm that does semantic based ranking of orders of matches in query of services. Similar to works by Noia in contrary to *open world semantics* of the underlying description logic semantics of OWL, **Grimm et al.** have proposed in [Grimm et al., 2006, Grimm & Hitzler, 2008] that an analysis to use *local closed-world reasoning*[4] to reason about service match-

[4] In traditional AI planning a closed world assumption is made with the meaning that if a literal does not exist and cannot be proved to be true in the current state of the world, its truth value in the current state is considered *false*, i.e., everything that is unknown to an agent is assumed to be false. In logic programming a corresponding epistemic approach to knowledge is called *negation as failure*. An incurred trouble with the closed-world assumption is that merely with the true and known literals one cannot express that new information and possibly true literal has been acquired when the state of the world changes.

ing and evaluate its applicability to matching. Two non-monotonic extensions to description logics: autoepistemic DLs and DLs with circumscription have been used to overcome traditional problem incurred with closed-world reasoning. In [Grimm et al., 2004] matching notion is further refined to take into consideration the variance between the human intuition of the service modeler and the formalism used to enable service discovery by proposing an approach to map the intuition into description logics constructs to incorporate it into reasoning. Service discovery and matching process thereby is based on service description as classes and instances. Variance in the matching process is described in two dimensions: *intended diversities* due to possibly many matched instances in the domain within a single world and *complete knowledge* due to matched instances in all possible worlds where ranking of the degree of matching is possible using formal DL inference.

Ian Horrocks et al. have proprose in [Li, 2004, Li & Horrocks, 2003a, Li & Horrocks, 2003b] an approach to formulate service capabilities as *service advertisements* which describes the functional and non-functional properties of a service using description logics constructs. Desired capability is formulated as query which is also based on description logics, one of the first approaches to do so, while matching algorithm observes the set based view of matching between concepts and relations in the query and in the advertisements that is conformed to the standard semantic matching model proposed in [Paolucci et al., 2002]. Matching is defined over the notion of compatibility of the set of queried concepts and those in the advertisement which is defined in terms of concept satisfiability based on set intersection between the queried set and the advertisement set.

Likewise interesting approach in semantic matching of services is published in [Hull et al., 2006] where services are described with description logics in a knowledge base comprising *TBox* and *ABox*. Service descriptions are formalized to tuples consisting of lists of pairs enumerating input parameters and their corresponding types as well as output value pairs with their corresponding types respectively. Services are consequently described in such enumerated conjunctive lists of tuples where instances in a corresponding *ABox* represent the instances of input and output lists characterizing the services. In a proposed matching formalization, a query of a certain service is likewise represented as a tuple containing the enumerated input type list, output value list and instances from ABox. Consequently service matching can be reduced to checking *containment* between two such *conjunctive queries* in description logics, one for the query and the other one for a matched service candidate representing in similar conjunctive tuples w.r.t. a *TBox*, which is a standard decidable reasoning task.

Finally the researchers of the group with **Di Sciascio et al.** have shown in works [Colucci et al., 2003a, Colucci et al., 2003b, Colucci et al., 2004, Colucci et al., 2005a] that semantic service discovery and matching can be tackled using a formalized approach of *concept contraction* and *concept abduction* which are specific extensions to standard description logics inference mechanism [Baader et al., 2007] to solve *concept covering* problem w.r.t. matching definition over intersection of sets of concepts as described in the semantic matching model in [Paolucci et al., 2002]. In a non-monotonic inference service, concept abduction and contraction represent, which extends concept subsumption and concept satisfiability respectively, can refinement in a knowledge and belief revision framework regarding logic based matching and query refinement which allows determining the quality of possible matches.

3.2 Service Composition Synthesis

We proceed with our survey on several well-defined techniques for service composition synthesis. We recapitulate the definition of the problem of service composition synthesis which states that it is concerned with synthesizing a distinct composite service that realizes the client request upon a certain set of complex

functionalities and behavior by appropriately coordinating available services. We review some of the existing approaches regarding the aspects of the following:

1. the way in which the client request is modeled;
2. the nature of the composition regarding the underlying interaction model and the way in which the modeled behavior of services and the world state interact;
3. the proposed architecture for coordination and orchestration in terms of possible support of dataflow and control flow.

A client request can be characterized according to its interaction model w.r.t. a service description, its degree of completeness and observability as described previously in section 3.1.2. A request is characterized as *monolithic* if it specifies that signature of the service that the client wants to realize in terms of input values, output data types and possibly preconditions and effects. Moreover client request can also be characterized in terms of the actions that the composite service will execute by specification of either *sequential* or *tree-based interaction* properties. From the perspective of the former one, a client specifies a set of temporal linear sequences of executable actions determined on the basis of properties expressed in the request without the client having to intervene the execution. The latter one specifies executions that are temporally representable in a tree form with the client obtaining choices of possible actions among which it can choose from and therefore tree-based interaction represents to certain degree at each point in execution a set of possible future states based on the executable actions to choose from. It is worth mentioning that client request that is specified as such do not necessarily share the same interaction model with the composite service.

Client specification can be either a complete or partial specification. The former denotes a deterministic single service with completely specified behavior that the client wants to realize while the latter denotes a set of partial services with any of which can realize the desire composite service, leaving a certain degree in the composite behavior unspecified and letting the composite service realizing it. Consequently it is interesting to study what kind of impact partial client specification will have on algorithmic realization for automatic service composition in the latter case.
While temporal sequential interaction is a simpler case when at each single step only one service executes sequentially. The tree-based interaction with possible partial client specification allows the possibility of interleaved concurrent executions among the component services which synthesize the composite service. The latter case is challenging in the sense that several services can be simultaneously active and concurrently executing. Also worth mentioning is the way how the component services behave in the latter case of partial specification when composite services can either execute regardless of all possible states of the component service at runtime or it must be realized by component services which execute actions to gather information to complement the underspecified composite service to satisfy certain conditional constraints such as guard conditions on certain state transitions during runtime.

3.2.1 Industrial Approaches

The **Business Process Execution Language (BPEL)** has constructs to allow combination of processes. The relevant **Web Services Business Process Execution Language (WS-BPEL)** [Alves et al., 2007] is one of the efforts in the industry to invent a solution for specification of process execution and combination for process oriented services. WS-BPEL is based on XML and specifies standard structures which are necessary for web services orchestration. Service composition is based on the modeling of *workflow*. Fundamental to the model is a set of *activities* that represent message exchange or intermediate result of processes. A process is conceptually defined to consist of a set of activities. Activities are structured with dedicated constructs such as sequential execution, switching for choices of execution, loop structure, etc. where variables are used

to hold messages and data and messages can be assigned identifying correlation to allow stateful conversation. The interaction model of WS-BPEL is largely monolithic because it incurs fixed completely specified and deterministic executions. Another drawback is the lack of semantics because services are described in WSDL.

Another previously mentioned effort is **Web Services Choreography Definition Language (WS-CDL)** [Kavantzas et al., 2005]. Composition in the choreography approach represents peer-based communication between services which are mutually controlled by the messages exchanged without resorting to any external instance for coordination. Therefore a WS-CDL specification is an agreement from an external perspective referring to the observable behavior of the coordinated services. As mentioned the underlying process model of WS-CDL is π-calculus defined by finite state transition systems using predefined communication channels.

Very few works exist which describe automatic service composition, i.e., in the sense of WS-BPEL and WS-CDL, to study how such syntactic specifications can be automatically generated given description of a desired target service (composite service) and a set of available component services. This deficiency still persists currently with the fact that WS-BPEL and similar solutions represent only semi-automatic or non-automatic mechanisms to compose services. Nevertheless, in the academia, many research efforts have proposed such automatic composition techniques which are theoretically based on related academic disciplines spanning across topics such as agent theory, automated theorem proving, artificial intelligence (AI) planning, etc. These techniques deal with monolithic as well as the sequential and tree-based interaction models and therefore represent far more desirable solution attempts to tackle the service composition problem. In the following, some of the most significant results are briefly reviewed.

3.2.2 Academic Approaches

McIlraith's Group

In the works [McIlraith & Son, 2001, McIlraith et al., 2001, McIlraith & Son, 2002] and with slightly different approach [Narayanan & McIlraith, 2002, Narayanan & McIlraith, 2003] by **McIlraith et al.**, they have presented a framework for automatic service composition that is based on the basic action theory of the situation calculus by **Reiter** [Reiter, 2001a] and AI planning. Service are represented as generic procedures in the agent-based GOLOG/ConGolog language with the corresponding information model expressed in an OWL-S service ontology. A user presents a request to the framework that is expressed as a GOLOG/ConGolog generic procedure with constraints and possible preferences. A user specification can be executed by an agent which internalizes the OWL-S service ontology and instantiates the services within the ontology. An agent takes described constraints and preferences into account and tries to execute the services. The GOLOG/ConGolog generic procedure of the composite service is actually associated with a situation tree with each node denoting a situation as the state of the service at a certain point in the execution.
Due to incomplete knowledge on the effects of execution of service actions, the agent executes knowledge gathering actions in an interleaved manner together with actual actions to update its belief state about the world in order to determine possible successor situation when a certain action is executed subsequently. Therefore the obtained trajectory of actions that the agent has executed so far includes interleaved world-altering actions, i.e., the actual actions corresponding to a service operation of a GOLOG/ConGolog generic procedure and knowledge gathering actions. It is possible that several such action trajectories can be obtained within the situation tree with each one of these representing a possible sequence of actions of the composite service to execute. It specifies the order that the generic procedures representing the component services can be instantiated and executed and represents acceptable sequence of actions in the sense of AI planning to satisfy a particular user goal. More recently **Sohrabi et al.** have shown in [Sohrabi et al., 2006] that user

preferences are customizable and can be formally incorporated into the generic procedures for reasoning and theorem proving tasks for web service composition based on McIlraith's approach. Ulterior approach to tackle the problem using planning has been shown in [Fadel & McIlraith, 2002, Baier & McIlraith, 2006a, Baier & McIlraith, 2006b, Baier et al., 2006, Sohrabi & McIlraith, 2008, Sohrabi et al., 2008] which are also worthwhile readings for comprehension of some of the relevant fundamental issues.

The Roman Group

In [Berardi et al., 2003a, Berardi et al., 2003b, Berardi et al., 2005c, Berardi et al., 2005d], a group of Roman researchers led by **De Giacomo, Calvanese et al.** have proposed a useful model that abstracts service behavior as finite state transition systems that are either deterministic or observably non-deterministic. Services are modeled as *execution trees* where they admit a representation as deterministic finite state machine. Services are observed as the executable actions (operations) which they offer. Composition is based on observable behavior of a (virtual) *target service* which is presented in a *client specification* and realized by suitably combining and coordinating a set of available component services. The composition is a synthesis of Mealy finite state transition system that requires full delegation of the actions of the target service to available services in a service community for execution. Each action in the target service is labeled with the component service to which it can be delegated. Each possible sequence of actions on the execution tree corresponds to possible interleaved sequence of actions of the available services. At each execution step the client has control of what action to execute next since it is presented choices of possible actions to execute by the target service.
While the behavior of a target service and the component services is expressed in an external schema, an internal schema is used to specify detailed delegation of individual actions to specific component services. A composition of the target service exists there exists a Mealy internal schema for the target service that is an Mealy finite state transition system which is expressed in a form of deterministic propositional dynamic logic (DPDL) [Fischer & Ladner, 1979, Ben-Ari et al., 1982] described in [Berardi et al., 2004b, Berardi et al., 2005c]. Checking existence of composition has EXPTIME complexity. Such a composition is reusable in the sense that it is a bottom up approach by client specification of a target service representing desired behavior and the synthesis attempts to assemble such desired behavior from available services. The interaction model of this approach is a tree-based branching model where each node in the execution tree denotes a choice point of the next possible actions.
A more recent variation of this approach adopts and utilizes the concept of *simulation* [Milner, 1971] to derived service composition synthesis based on simulation preorder between finite state transitions systems [Berardi et al., 2008, Sardina et al., 2007, Sardina et al., 2008, Patrizi & Giacomo, 2009]. It has favorable runtime complexity characteristics and is more tolerant in failure situations [Sardina et al., 2008]. It is worth mentioned that non-deterministic behavior due to uncertainty of outcomes of action executions has led the researchers to come up with a solution to deal with non-determinism [Berardi et al., 2006a, Berardi et al., 2006b, Berardi et al., 2004c] with underspecified client specification, as a loosely specified target service transition system, which enables the target services to incorporate an implementation that allows it to take decisions on whether to execute actions on its own or to delegate them in order to cope with uncertainty. Also interesting is the fact that there has been an initial attempt by the group to combine service behavioral description with handling of dataflow in the COLOMBO framework [Berardi et al., 2005a].

Hull and Su's Group

Hull, Su et al. have shown in [Bultan et al., 2003, Hull et al., 2003, Gerede et al., 2004, Hull, 2005] that services exchange messages according to a certain communication topology expressed in communication channels that are buffered queues among services. A sequence of exchanged messages using the dedicated,

bounded channel is referred to as specifciation of a *conversation* and service behavior is modeled as Mealy finite state machine which Hull calls a mediator for coordinating the set of *conversations* among peer services. The composition of services is a synthesis problem with a set inputs consisting of the desired global behavior expressed as a set of all desired conversations in a certain temporal logic and a composition infrastructure consisting of a set of services and messages as well as a set of associated channels. Output of the composition synthesis is the specification of the Mealy machine of the services such that their conversations are compliant with the expressed specification of desired conversation. The conceptual interaction model underlying this approach is a sequential model because the Mealy composition focuses on linearly ordered sequences of messages from a global perspective and there is no account from the client perspective of the Mealy automaton about decision of possible messages to send at each step for the next step. Therefore Hull's approach is in this regard different to the Roman approach though both proposals attempt to use finite state machines to model behavior.

Miscellaneous Other Research Efforts

In the AI planning approach proposed in [Pistore et al., 2004, Pistore et al., 2005a, Pistore et al., 2005b] by **Traverso et al.**, they have proposed a composition algorithm which takes in input a set of partially specified services, modeled as non-deterministic finite state machines and a client goal specified in a specific planning domain language. The algorithm returns a plan that specifies how to coordinate the execution of concurrent services in order to realize the goal. The plan can serve to monitor the composition in the sense that available services behave consistently with their specifications. Due to the presence of non-determinism and partial observability of states, the search space in planning is quite large even for simple services with few states. An approach is proposed to overcome the problem by using symbolic modeling checking techniques and efficient heuristics to prune search space and avoid undirected searches during the generation of the plan.

Knoblock et al. have proposed to use data integration techniques to dynamically compose atomic services of some data sources [Thakkar et al., 2002, Thakkar et al., 2003, Thakkar et al., 2004, Knoblock et al., 2005]. Their composition algorithm takes in the set of data source fed services as input and a user query expressed in terms of user input and requested outputs. An output is a composite service that is able to execute an integration plan for a *template* query such that all input values can be answered by the composite service. This approach adopts a mediator concept by enabling the mediator to construct an integration plan consisting of a sequence of source queries and binding the template source queries with concrete values. The integration plan is generated with a forward chaining planning algorithm. The interaction model is a monolithic model with dataflow taken into account based on abstraction of services associated with their data sources which are represented as views on a relational database.

An interesting approach to combine AI planning and well-known description logic inference mechanisms for service composition has been proposed by **Sirin et al.** in [Sirin et al., 2003a, Sirin et al., 2003b, Sirin et al., 2004a, Sirin et al., 2004c, Sirin et al., 2005a, Sirin et al., 2005b, Sirin, 2006]. Sirin's approach combines hierarchical task network (HTN) planning[5] [Russell & Norvig, 2002] and OWL-S service ontology descriptions which are based on semantics of description logics (DL) to generate composition plan. The proposed variant formalism HTN-DL combines HTN planning and DL to provide sufficient expressiveness based on DL and efficiency of HTN planning (SHOP2) systems to solve service composition problem in semantical rich way. This approach provides a translation algorithm to transform OWL-S service descriptions available as templates to HTN-DL to encode the control constructs used to describe the control flow of web service in an HTN-DL planning domain formalism. It also provides a semantics for OWL-S processes which

[5]HTN planning is a useful automated planning approach to break down complex planning task into a hierarchical network of tasks and support conditional and decision during planning.

is compatible with the originally proposed situation calculus based semantics of OWL-S. Some optimization techniques for DL reasoning which target nominals and large number of individuals have bee proposed to allow the HTN-DL planner to efficiently reason with OWL-DL ontologies during planning.

Some other grounded discussions on occasionally mentioned topics regarding service composition, e.g. nature of service composition, complexity issues are those given by **Lécué et al.** in [Lécué & Leger, 2006, Lécué & Delteil, 2007], by **Di Sciacio et al.** in [Colucci et al., 2005b, Noia et al., 2005, Ragone et al., 2007] and by **Shen et al.** in [Shen & Su, 2007a, Shen & Su, 2007b], whereby these approaches will not be exhaustively explained and readers can consult these publications directly.

3.3 Comparative Juxtapositions

Based on the service modeling characterization aspects of: *(i) interaction model*, *(ii) controllability and state observability* and *(iii) dataflow handling ability* described in section 3.1.1, the following tables show comparatively the similarities and differences of the different approaches in both service description and modeling (shown in table 3.1) and service composition synthesis (shown in tables 3.2 and 3.3) respectively. Columns are labeled with the originating group of researchers for a specific proposed approach, references can be consulted in the previous sections where each approach is explained with source hints to literature and therefore are not otherwise repeated in the tables.

	WSDL	**WSCL & WSCI**	**WS-CDL**	**WS-BPEL**	**OWL-S**	**WSMO**	**Ambite et al.**
Interaction Model	monolithic	tree-based	sequential	sequential	*ServiceProfile* monolithic; *ServiceModel* sequential	sequential	monolithic
Completeness & Observability of States & Exported Behavior	N.A.	full	full	full	N.A. / not addressed / no support	N.A.	N.A.
Support Dataflow	yes (XML document instances)	yes (XML document instances)	yes	yes	limited	no	yes (views over relational database/-datasource)

	Hull et al.	**Deutsch et al.**	**De Giacomo et al.**	**Pistore et al.**	**McIlraith et al.**	**Di Sciascio et al.**	**Sirin et al.** [Sirin et al., 2004b]
Interaction Model	sequential	tree-based	tree-based	tree-based	tree-based	monolithic	sequential
Completeness & Observability of States & Exported Behavior	full	full	full and partial observability supported	partial	N.A.	N.A. / not addressed / no support	partial
Support Dataflow	no	yes (relational schema)	partial (support in COLOMBO)	no	partial (via GOLOG/-ConGolog parameters)	no	no

Table 3.1: Comparative Juxtaposition of Service Description and Modeling Approaches

	WS-CDL	WS-BPEL	OWL-S	WSMO	McIlraith et al.	De Giacomo et al.	Hull et al.
Client Specification Interaction Model	N.A.	N.A.	monolithic and sequential	N.A.	sequential	tree-based	sequential
Completeness & Observability in Client Specification	full	full	full	N.A.	partial	full and partial	full
Type of Composition Execution	sequential	sequential	sequential, conditional	sequential, conditional	sequential, conditional	sequential, concurrent	sequential, concurrent
Orchestration or Coordination Model	mediation	mediation	N.A.	mediation	mediation	mediation and delegation	peer to peer
Type of Composition	manual	manual	semi-automatic	semi-automatic	automatic	automatic	automatic
Support Dataflow	yes	yes	limited	no	partial (via GOLOG/-ConGolog parameters)	no	no

Table 3.2: Comparative Juxtaposition of Service Composition Synthesis Approaches (Part 1)

	Traverso et al.	Knoblock et al.	Sirin et al.	Lécué et al.	Di Sciacio et al.	Shen et al.
Client Specification Interaction Model	sequential	monolithic	Sequential	N.A.	N.A.	sequential
Completeness & Observability in Client Specification	full	N.A.	partial	N.A.	N.A.	full
Type of Composition Execution	conditional, concurrent	sequential	sequential, conditional	N.A.	N.A.	sequential, concurrent
Orchestration or Coordination Model	mediation	mediation	mediation	N.A.	mediation	peer to peer
Type of Composition	automatic	automatic	automatic	manual or semi-automatic	automatic	automatic
Support Dataflow	no	yes	no	no	no	yes

Table 3.3: Comparative Juxtaposition of Service Composition Synthesis Approaches (Part 2)

4 Formalisms for Service Matching and Service Composition

In the landscape of process oriented enterprise services, deciding on which enterprise service to select to accomplish a task is often a challenging issue. This is partly due to a vast number of available enterprise services on one side which expose a backend enterprise resource planning system such as the SAP services; and on the other side the lack of effective description of service capabilities and appropriate discovery mechanisms to find services. Beside terse textual documentations there is often no other sources to know about their functionalities. Unintelligent keyword based service search is provided to query for service; inflexible configuration of service registries and administrative overhead further limit efficient discovery and usage of enterprise services. Since there exists very limited semantically rich description of service capabilities, contemporary interaction with enterprise services is still characterized by either manually intervention-centric search and selection or rigid machine-to-machine communication with high coupling among processes and their exposed services, making flexible reconfiguration difficult for them.

4.1 Semantic Service Matching and Discovery

We have reviewed some of the important approach proposals in chapter 3 concerning service modeling, service matching and service composition. A first step towards better interoperability for existing enterprise services is an improvement of ability to locate the 'right' services from potentially a large bunch of similar or functionally related services. Potentially they can all offer help toward the solution of a problem though we certainly do not want to use or test each one of them before use. Since traditional syntactic web services languages and discovery mechanisms do not support intelligent selection per se, the task of location of enterprise services has been to a large extent based on enterprise registries such as UDDI which has proved to be not very scalable. It fails to catch on probably also due to high administrative overhead and unintelligent search query answering merely based on categorized tags and keyword based search.

Deciding and evaluating quality of matches and degree of appropriateness of matches in terms of variance between available functionality and requested functionality using traditional syntactic search and matching techniques are outside the representation capabilities of UDDI registries. Thus a more flexible and intelligent type of registries should be based on rich semantic description formalisms to abstract service capabilities into shared knowledge collections such as a set of service ontologies reflecting in an expressive way the actual service functionality. Rich semantic description can be reasoned about using decidable logical inference mechanisms using for instance logic reasoners to enable more precise query answering, satisfying functionality attributes constraints, user preferences or even service quality attributes. A declarative approach of service capability advertisement represents a key development towards more intelligent and automated enterprise service discovery, matching and selection.

4.1.1 Formal Abstract Service Description

From a pure syntactic perspective, we conceive an enterprise service as a *generic web procedure* based on some underlying process configuration and processing viewed as a black box, abstracting away from the conversational state with client and concrete details of the implementation. Each enterprise service as such can be represented by a concept name S denoting the service, a processing function δ, a finite set of internal service states B_{int} representing the different possible states within the processing of a process oriented service, a list $\vec{P}$ consisting of n finite input parameters $P = \{p_1, \ldots, p_n\}$ where p_i denotes one argument component in the list and $n \in \mathbb{N}$. A service performs an operation that is intrinsic to its operation semantic which we assume that service will success without any further prerequisite. After an operation finishes, depending on the characteristic of the service, it can return a value. This can be represented with a list $\vec{R}$ which can either be empty (if no value is returned or the service is not configured to return anything) or it can consist of a finite set of return values[1], i.e., $R = \{\emptyset \mid \{r_1, \ldots, r_m\}\}$. The variable n is the *arity of parameter list* which denotes the size of the list.
Enterprise services must be deployed by creating an evocable unit of the service from the service definition S. The evocable unit is called an *instance* of S and is denoted by s_I. An instance s is assigned a copy of the processing function δ_s with the corresponding set of internal states B_{int_s}. The instance accepts a list of parameters containing *argument objects* which are instances of the corresponding argument components of the parameter list in the service definition, denoted by $\vec{P}_s = [a_1 : p_1, \ldots, a_n : p_n]$ where a_i is an instance of argument of type p_i. The instance is allocated the necessary resources to finish processing and can return a list $\vec{R}_s = [b_1 : r_1, \ldots, b_m : r_m]$ which is either empty or contains a finite set of returned objects corresponding to the types of the components in the return list of the service definition.

Definition 4.1.1 (**Abstract Enterprise Service Syntax**)**.** An enterprise service can be defined syntactically as follows:

$$S^n(\vec{P}) : \delta(\vec{P}) \times B_{int} \rightarrow \vec{R}$$

where

- S is the name of the service and S^n is a service with an arity n of input parameters;
- $\vec{P}$ denotes the input parameters in a list structure. The size of the list is n;
- δ denotes a processing function containing the predefined business logic underlying the service. $\delta(\vec{P})$ denotes this function with the input parameters;
- B_{int} denotes a finite set of internal service states representing the different processing states within the services;
- $\vec{R}$ denotes the result list containing output of the service.

The service definition $S^n(\vec{P})$ is a function which maps the set of input parameters to a set of (possibly empty) output values. This definition is called a *service signature*.

□

Definition 4.1.2 (**Abstract Enterprise Service Instance Syntax**)**.** An enterprise service instance is an object serving a request and is created from a service definition S by proper resource allocation and assignment. It is defined syntactically as follows:

$$s^n.S^n(\vec{P}_s) : \delta_s(\vec{P}_s) \times B_{int_s} \rightarrow \vec{R}_s$$

where

[1] One can think of this as a result list where the returned value is a list consisting of individual output objects.

- s^n is the identifier of the instances with the arity n of input parameters. The instance belongs to the type of service S^n on which the instance is created and the notation s_i^n is used to uniquely identify an instance among others;
- $\vec{P}_s$ denotes the input parameter list with size n that belongs to the instance;
- δ_s denotes a copy of processing function that is assigned to the instance;
- B_{int_s} denotes the finite set of internal states belonging to the instance;
- $\vec{R}_s$ denotes the result list of the instance.

Being an object, an instance has a life cycle. It can be created, deactivated if not needed, reactivated to serve request, pooled or destroyed and evicted from memory by the system that deploys the instance. We call the above instance definition an *instance signature*.

□

A community of enterprise services C is defined as a finite set of service definitions as described in definition (4.1.1). An instance community C_s is the corresponding finite set of available and activated instances based on the service definitions in C that are ready to serve requests. While the size of C is usually fixed and does not subject to change a lot provided that no definition is removed from the community frequently for instance because of maintenance or removal of outdated services; the size of an instance community C_s do change quite often because instances have comparably shorter life cycle.

4.1.2 Formal Abstract Service Matching

Given a finite set of enterprise services in a community C of size m containing the following service descriptions and a query $Q := S^l_{query}(\vec{P}) \rightarrow \vec{R}$ searching for a service with the signature as shown:

$$
\begin{array}{lll}
S_1^l(p_1, \ldots, p_l) & \rightarrow \vec{R} & \text{(service 1)} \\
S_2^n(p_1, \ldots, p_n) & \rightarrow \vec{R} & \text{(service 2)} \\
\vdots & \vdots & \\
S_m^k(p_1, \ldots, p_k) & \rightarrow \vec{R} & \text{(service m)} \\
 & & \\
S^l(p_1, \ldots, p_l) & \rightarrow \vec{R} & \text{(query)}
\end{array}
$$

The services in C are numbered by subscripts. The following fundamental question is essential to any service search and discovery system: how to select an appropriate match that can satisfy the query?
The answer w.r.t. definition of the abstract service syntax is not difficult: a match can be determined by an algorithm based on the notion of matching *service signature* by performing the following steps:

1. first it tries to match the query service name against a name S_i in C, if no match is found it aborts;
2. then it matches the arity in the query with that of the services in C, if there is no match it aborts;
3. with a match of arity, it tries to match the parameter types of the argument components in the candidate service in exactly the order they appear, if type does not match, selecting the service for invocation may cause failure;

4. finally it tries to match the type of $\vec{R}$ in the query against the one in the candidate service. Notice that a mismatch in this step will not necessarily cause failure since the return value may still be ignored provided that the previous matching criteria are met. The client can still invoke the service and choose to discard the return value[2] while unmatched input parameters cause immediate failure on invocation of the service.

If there are multiple matching candidates with slightly different signature, for instance every criteria matches except that the return value differs in terms of type generality, these candidates can be returned in an answer set for choice.

On the instance level, once service instances are involved the above matching criteria must be extended in order to safely select service instance to answer a query. This is because the type membership of the parameter objects must be accounted for polymorphically, i.e., a parameter object does not only belong to its direct type but also a more general type from which it derives. This is where the issues of *covariance* and *contravariance* [Meyer, 1990, Castagna, 1994] of types regardings input parameter and return type of the service can influence a decision of match. The *contravariance rule* captures the *substitutivity* of types, i.e., subtyping relation that specifies which generalized set of types can safely replace another set of given types in every context. In juxtaposition, a *covariance rule* characterizes the *specialization* of type which specifies the creation or referencing of more special types to replace the general ones in a more specific context. Given an instance community C_s consisting of:

$$
\begin{array}{lll}
S^l_A(p_1,\ldots,p_l) & \rightarrow \vec{R} & \text{(service A definition)} \\
 & & \\
s^l_1.S^l_A(a_1:p_1,\ldots,a_l:p_l) & \rightarrow \vec{R}_s = [o_1:r_1,\ldots,o_n:r_n] & \text{(instance 1 of service A)} \\
s^l_2.S^l_A(a_1:p_1,\ldots,a_l:p_l) & \rightarrow \vec{R}_s = [o_1:r_1,\ldots,o_n:r_n] & \text{(instance 2 of service A)} \\
s^l_3.S^l_A(a_1:p_1,\ldots,a_l:p_l) & \rightarrow \vec{R}_s = [o_1:r_1,\ldots,o_n:r_n] & \text{(instance 3 of service A)} \\
\vdots & \vdots & \\
s^l_x.S^l_A(a_1:p_1,\ldots,a_l:p_l) & \rightarrow \vec{R}_s = [o_1:r_1,\ldots,o_n:r_n] & \text{(instance x of service A)}
\end{array}
$$

$$
\begin{array}{lll}
S^m_B(p_1,\ldots,p_m) & \rightarrow \vec{R} & \text{(service B definition)} \\
 & & \\
s^m_1.S^m_B(a_1:p_1,\ldots,a_m:p_m) & \rightarrow \vec{R}_s = [o_1:r_1,\ldots,o_n:r_n] & \text{(instance 1 of service B)} \\
s^m_2.S^m_B(a_1:p_1,\ldots,a_m:p_m) & \rightarrow \vec{R}_s = [o_1:r_1,\ldots,o_n:r_n] & \text{(instance 2 of service B)} \\
\vdots & \vdots & \\
s^m_y.S^m_B(a_1:p_1,\ldots,a_m:p_m) & \rightarrow \vec{R}_s = [o_1:r_1,\ldots,o_n:r_n] & \text{(instance y of service B)}
\end{array}
$$

$$
\begin{array}{lll}
S^n_C(p_1,\ldots,p_n) & \rightarrow \vec{R} & \text{(service C definition)} \\
 & & \\
s^n_1.S^n_C(a_1:p_1,\ldots,a_n:p_n) & \rightarrow \vec{R}_s = [o_1:r_1,\ldots,o_n:r_n] & \text{(instance 1 of service B)} \\
\vdots & \vdots & \\
s^n_z.S^n_C(a_1:p_1,\ldots,a_n:p_n) & \rightarrow \vec{R}_s = [o_1:r_1,\ldots,o_n:r_n] & \text{(instance z of service B)} \\
\vdots & \vdots &
\end{array}
$$

[2] Unmatched result value is not a necessary condition of failure, therefore in such case the candidate service can still be selected to answer the service query.

and a query

$$Q := s^n_{query}.S^n_{query}(q_1 : p_1, \ldots, q_n : p_n) \;\; \rightarrow \;\; \vec{R}_s = [t_1 : r_1, \ldots, t_n : r_n]$$

to select a matching service instance from an instance community. A matching algorithm, in addition to the previously described steps, is required to take care of these following aspects:

1. for the size (arity) of the parameter list of the query, the matching algorithm can try to find a set of candidate service instances with matching arity and filter out those mismatch ones in order to reduce the size of search space;

2. for each input parameter object $q_1, \ldots, q_n$ of the query, it iteratively checks

   $$(\forall q_i \in \vec{P}_s \wedge \vec{P}_s \neq \emptyset) \bigwedge_{i=1}^{n} (q_i \sqsubseteq a_i) \vee (q_i \sqsupseteq a_i),$$

   i.e., whether the query input parameter objects belong type-wise to the corresponding parameter objects of candidate service instances in the given order they appear in the list. The notations "$\sqsubseteq$" and "$\sqsupseteq$" denote instance subsumption relations where $a \sqsubseteq b$ denotes that the object a is subsumed by b such that b belongs to the generalized type to which a also belongs. The other notation "$\sqsupseteq$" indicates that the subsumption goes in the other direction. If there exists one pair of objects with

   $$(q_i \sqsubseteq a_i \equiv \bot) \wedge (q_i \sqsupseteq a_i \equiv \bot),$$

   i.e., subsumption relation does not exist in either direction and the check fails for both, then the candidate instance must be rejected due to mismatch;

3. the specification of input parameters for matching on the instance level must conform to the *covariance rule* of parameters which states that the type of input parameter objects of candidate service instances represents a more specific and stringent type-checking than the query parameter objects in order to guarantee an appropriate degree of matching and selection precision;

4. for each output objects in the result list $\vec{R}_s = [t_1, \ldots, t_n]$, the matching algorithm iteratively checks

   $$(\forall t_i \in \vec{R}_s) \bigwedge_{i=1}^{n} t_i \sqsubseteq o_i$$

   which means that each possible output object of the query is subsumed by the output object of the candidate service instance. If it is found that

   $$t_i \sqsubseteq o_i \equiv \bot,$$

   the candidate service instance must not be returned as answer to the query;

5. the specification of output objects of the service instances conform to the *contravariance rule*[3] stating that they should be more general than those in the query in order to guarantee that they can be replaced by the more specific output objects of the query instance.

With this we have had a survey of the fundamentals of service matching from a syntactic perspective, next we investigate rich description of service capability with semantic annotation for matching purpose and define how semantic matches are evaluated.

[3]Hence the subsumption relation is checked only in one direction but not both in the last point.

4.1.3 Notion of Semantic Service Description and Matching

We have shown essential service description and matching issues based on a rather strict view of generic procedure to build an abstract model of matching of enterprise services. However no description of the capabilities of services has been made. We introduce semantic matching to answer query on functionality of services.

Description Logics

Description logics [Baader et al., 2007] (DL) are a well-known family of knowledge representation formalisms. They are based on notion the concepts such as unary predicates, concept classes and roles such as binary relations which are mainly characterized by constructors that allow complex concepts and roles to be built from atomic concepts and roles. DL Concepts and roles are defined in a *knowledge base* denoted with $\mathcal{KB}$ consisting of a *TBox* which is a shorthand for terminology box and an *ABox*, which is a shorthand for assertion box, containing instances of concepts and roles. A DL knowledge base comprises a TBox, an ABox and a reasoning facility to infer new knowledge from existing knowledge using logical inference mechanisms as well as performing a series of logical operations on the knowledge base such as concept classification, subsumption proving and checking satisfiability of concepts w.r.t. a TBox and an ABox. A DL *reasoner* such as RACER [Haarslev & Möller, 2001b, Haarslev & Möller, 2003, Haarslev & Möller, 2001a, Möller & Haarslev, 2003] and RacerPro that are efficient and use DL *tableau method*[4] [Baader & Sattler, 2001, Ortiz et al., 2006a] for deciding logical inference. Tableau method is a type of logical decision procedure in logical proof theory that can be used to carry out the previously mentioned logical operations very efficiently. Atomic concepts and roles are built from DL constructors, for instance, constructors for the DL lanugage $\mathcal{ALC}$ [Baader et al., 2007] with A, C and D as atomic concept names are as follows:

$$
\begin{array}{rcl}
C, D & \rightarrow & A \text{ (atomic concept)} \\
 & & \top \text{ (universal concept, always satisfiable)} \\
 & & \bot \text{ (bottom concept, never satisfiable)} \\
 & & \neg A \text{ (atomic negation of concept)} \\
 & & C \sqcap D \text{ (intersection of concepts)} \\
 & & \forall R.C \text{ (universal role restriction)} \\
 & & \exists R.\top \text{ (limited existential quantification)}
\end{array}
$$

and for more expressive semantic description, the variant $\mathcal{SHIQ}(D)$ [Baader et al., 2007] can be used:

$$
\begin{array}{ll}
\exists R.C & \text{(full existential role restriction)} \\
\neg C & \text{(negation of arbitrary concepts)} \\
\leq n\, R & \text{(utmost cardinality restriction on role)} \\
\geq n\, R & \text{(at least cardinality restriction on role)} \\
= n\, R & \text{(exact cardinality restriction on role)} \\
\leq n\, R.C & \text{(qualified utmost cardinality restriction)} \\
\geq n\, R.C & \text{(qualified at least cardinality restriction)} \\
= n\, R.C & \text{(qualified exact cardinality restriction)}
\end{array}
$$

Formal semantic of DL is characterized by an *interpretation* $\mathcal{I}$ that consists of a non-empty set $\triangle^{\mathcal{I}}$ representing the domain of interpretation. and an interpretation function which assigns to every atomic concept A a set $A^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}}$. We interpretation of more complex concepts is built from the interpretation of atomic concepts with the appropriate logical connectives,

[4] Tableau method is also known as tableau algorithm.

e.g., $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$ and $(\exists R.\top)^{\mathcal{I}} = \left\{a \in \triangle^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}}\right\}$. We say that two atomic concepts are equivalent, i.e., *concept equivalence* is defined as

$$C \equiv D \doteq (\forall \mathcal{I}).(C^{\mathcal{I}} = D^{\mathcal{I}}).$$

In a DL knowledge base, there are a set of terminological axioms that assert how concepts and roles relate to each other with the form: $C \sqsubseteq D$ for concepts and $R \sqsubseteq R'$ for roles. It specifies that a DL concept D is more general than another one C in which case the more general concept *subsumes* the more specific one. Axioms that denote this type of relation are called subsumption axioms. Subsumption relations contribute to **classification** of concepts within the knowledge base $\mathcal{KB}$. The set of classified concepts form the set of *general concept inclusion* (GCI) axioms in the *TBox*.

Service Capability Description with DL

In order to advertise the capability of an enterprise service, we resort to axiomatizing the functionality and other non-functional properties to express them in terms of general concept inclusions and concepts equivalence in a *TBox* which resembles an ontology for the service functionality. The point is that *TBox* reasoning is more effective than *ABox* reasoning due to known reasons [Hustadt et al., 2005, Calvanese et al., 2006, Calvanese et al., 2007] and most of the efficient logical inferential and decision algorithms are optimized for *TBox* reasoning. We do not limit ourselves in the specific semantic language such as OWL or other syntax in expressing service capability, instead we device a general ontology using GCIs which are able to describe a natural hierarchy of relationships between concepts. For instance in DL we define service profile for the description of capability:

$$\begin{array}{lcl}
enterpriseServiceProfile & \sqsubseteq & \top \\
enterpriseServiceParameters & \sqsubseteq & \top \\
serviceFunctionalityProperty & \sqsubseteq & enterpriseServiceProfile \\
serviceNonFunctionalProperty & \sqsubseteq & enterpriseServiceProfile \\
serviceQuery & \sqsubseteq & enterpriseServiceProfile \\
serviceFunctionalityQuery & \sqsubseteq & enterpriseServiceProfile \\
serviceNonFunctionalQuery & \sqsubseteq & enterpriseServiceProfile \\
inputParameter & \sqsubseteq & enterpriseServiceParameters \\
outputParameter & \sqsubseteq & enterpriseServiceParameters \\
\vdots & & \vdots
\end{array}$$

For a typical enterprise procurement service for instance the input parameters and output can be axiomatized inexhaustibly using GCIs[5]:

$$\begin{array}{lcl}
requestedBy & \sqsubseteq & inputParameter \\
identifiedBy & \sqsubseteq & inputParameter \\
hasCustomerId & \sqsubseteq & inputParameter \\
requestedProduct & \sqsubseteq & inputParameter \\
requestedQuantity & \sqsubseteq & inputParameter \\
usedPayment & \sqsubseteq & inputParameter \\
delivery & \sqsubseteq & inputParameter \\
deliveryDate & \sqsubseteq & inputParameter \\
\vdots & & \vdots \\
deliveredProduct & \sqsubseteq & outputParameter \\
\vdots & & \vdots
\end{array}$$

[5]GCI is a shorthand for general concept inclusion in description logics.

and use a profile to advertise the functionality:

$$
\begin{aligned}
enterpriseProcurementServiceProfile \quad &\sqsubseteq \quad enterpriseServiceProfile \sqcap \\
&\quad (=_1\ requestedBy.ServiceClient) \sqcap \\
&\quad (=_1\ identifiedBy.CustomerIdNumber) \sqcap \\
&\quad (=_1\ hasCustomerId.Integer) \sqcap \\
&\quad (=_1\ requestedProduct.Product) \sqcap \\
&\quad (=_1\ requestedQuantity.Integer) \sqcap \\
&\quad (=_1\ usedPayment.PaymentMethod) \sqcap \\
&\quad (=_1\ minDeliver.Integer) \sqcap \\
&\quad (=_1\ delivery.Location) \sqcap \\
&\quad (=_1\ deliveryDate.Date) \\
Product \quad &\sqsubseteq \quad (=_1\ hasName.ProductName) \sqcap \\
&\quad (=_1\ hasDescription.Description) \sqcap \\
&\quad (=_1\ hasPrice.RealNumber) \sqcap \\
&\quad (=_1\ hasStockQuantity.Integer) \\
PaymentMethod \quad &\sqsubseteq \quad (=_1\ hasName.MethodName) \sqcap \\
&\quad (=_1\ hasType.PaymentType) \sqcap \\
&\quad (=_1\ hasCreditLevel.RealNumber) \sqcap \\
&\quad (=_1\ hasCreditor.Person) \sqcap \\
&\quad (=_1\ hasDebtor.Organization) \\
\vdots \quad &\quad \vdots
\end{aligned}
$$

For instance a certain enterprise procurement service instance can be described as:

$$
\begin{aligned}
instancePSAdvertisement \quad \equiv \quad & enterpriseProcurementServiceProfile \ \sqcap \\
& (\forall requestedBy.(ServiceClient \sqcap \\
& \quad \forall hasName.\{SAP\})) \ \sqcap \\
& (\forall identifiedBy.(CustomerIdNumber \sqcap \\
& \quad \geq_{0001}\ hasCustomerId)) \ \sqcap \\
& (\forall requestedProduct.(Product \sqcap \\
& \quad \forall hasName.\{Bosch\,drill\,spare\,part\} \sqcap \\
& \quad \forall hasDescription.\{diamond\,drill\,head\} \sqcap \\
& \quad \geq_{2500}\ hasPrice \sqcap \\
& \quad \geq_{5000}\ hasStockQuantity)) \ \sqcap \\
& (\geq_{200}\ requestedQuantity) \ \sqcap \\
& (\forall usedPayment.(PaymentMethod \sqcap \\
& \quad \forall hasName.\{credit\} \sqcap \\
& \quad \forall hasType.\{Visa\} \sqcap \\
& \quad \geq_{24000.00}\ hasCreditLevel \sqcap \\
& \quad \exists hasCreditor.\{Bob\} \sqcap \\
& \quad \exists hasDebtor.\{Citibank\})) \ \sqcap \\
& (\forall delivery.(Delivery \sqcap \\
& \quad \leq_{2010.02.01}\ deliveryDate \sqcap \\
& \quad deliveryLocation.Boston))
\end{aligned}
$$

We notice that the above service advertisement is for a single instance within an instance community while other instances can exist simultaneously in the community with other functionalities which are either similar

to the current instance or quite different from it. An example query searching for a procurement service instance is shown in the following:

$$
\begin{aligned}
serviceQuery1 \quad \equiv \quad & enterpriseProcurementServiceProfile \sqcap \\
& (\forall requestedProduct.(Product \sqcap \\
& \quad \forall hasName.\{Bosch\,drill\,spare\,part\} \sqcap \\
& \quad \leq_{8500}\ hasPrice \sqcap \\
& \quad \geq_{3000}\ hasStockQuantity)) \sqcap \\
& (\forall usedPayment.(PaymentMethod \sqcap \\
& \quad \forall hasName.\{credit\} \sqcap \\
& \quad \geq_{20000.00}\ hasCreditLevel)) \sqcap \\
& (\forall delivery.(Delivery \sqcap \\
& \quad \leq_{2010.04.01}\ deliveryDate))
\end{aligned}
$$

Query Matching

A query can be formulated in a similar manner by a DL statement expressing a set of desired functionalities. A DL reasoner is used to check for satisfiability of the query against a set of service capability advertisements with standard known inference techniques to decide on matches. One can conceive the service instance advertisements as semantic axiomatizations of an application domain which expose the capability, functionality and perhaps non-functional properties in semantic interpretable statements. A query from the DL point of view is almost identical to the service instance advertisements since a query explicitly formulates a desired set of functionalities and non-functional properties regarding a service that is searched for. This requested set of functionalities and properties can either:

- be satisfied by one or more declared descriptions of instances that are available in the community in which case the capability set of the query can be viewed as a subset of the capability of the available instances;
- the query is unsatisfiable because there exists no single set of service functionalities and properties declared in the community which is either equivalent or partially coincide with the requested set in the query. One has to be careful with terminology since a real mismatch is only determined by the fact that the requested set and the declared set of functionalities are mutually disjoint. Otherwise it is still possible that some service instances can partially satisfy the request set of functionalities.

Unless the query is unsatisfiable due to mutual disjointness of the query set and the advertisement set, an answer can still be returned which incorporates the set of compatible or partially compatible service instances.

Definition 4.1.3 (Service Capability Query Matching). Let Ψ be the set of all service advertisements describing the capability of service instances semantically using concepts and roles of description logics which are stored in persistent repository of an instance community. Given a query Q with the set of desired requirements on functional and non-functional properties denoted with $\mathcal{F}_Q$ and a set of service advertisements denoted with $\mathcal{F}_A$ which represent instances that can potentially satisfy the query. We define a query matching function $Match$ such that $Match : Match(\mathcal{F}_Q) \rightarrow \mathcal{F}_A$. The matching function has the following semantic:

$$Match(\mathcal{F}_Q) \stackrel{def}{=} \{\mathcal{F}_A \subseteq \Psi \,|\, compatible(\mathcal{F}_A, \mathcal{F}_Q)\}$$

where the compatibility of two concepts C and D from the sets $\mathcal{F}_Q$ and $\mathcal{F}_A$ respectively is defined as

$$compatible(C, D) \doteq \neg(C \sqcap D \sqsubseteq \bot)$$

i.e., their intersection is not the emptyset $(\forall \mathcal{I} \in \triangle^I).\mathcal{F}_Q^I \cap \mathcal{F}_A^I \neq \{\emptyset\}$ or

$$(\forall C \in \mathcal{F}_Q \wedge \forall D \in \mathcal{F}_A).\{\neg(C \sqcap D \sqsubseteq \bot)\}$$

and thus the query set can be satisfied.

□

4.1.4 Deciding Matching Level of Services

Matching decision from the DL perspective can be reduced to standard reasoning problem of answering *conjunctive query* [Hull et al., 2006]. For instance, there is a service query formulated as a *conjunctive query* with $q(\vec{c}, \vec{r}) \doteq [qterm_1(\vec{c}, \vec{r}) \wedge \cdots \wedge qterm_n(\vec{c}, \vec{r})]$ where $\vec{c}$ and $\vec{r}$ denotes the vectors of concepts and roles as query parameters w.r.t. a knowledge base $\mathcal{KB}$ comprising *TBox* $\mathcal{T}$ and *ABox* $\mathcal{A}$. Given $\mathcal{KB}$ and a set of individuals $\vec{\mathcal{I}} = \langle i_1, \ldots, i_m \rangle$ in $\mathcal{KB}$, the answer to the conjunctive query $q(\vec{c}, \vec{r})$ is defined as:

$$(\exists \vec{a} \in \vec{\mathcal{I}}).q(\mathcal{KB}) \wedge \mathcal{KB} \models \exists \vec{y}.(qterm_1(\vec{a}, \vec{y}) \wedge \cdots \wedge qterm_n(\vec{a}, \vec{y}))$$

Conjunctive query answering is decidable [Calvanese et al., 2008a] with tractable computational complexity upper bounds [Calvanese et al., 2006, Ortiz et al., 2006a, Calvanese et al., 2008a, Ortiz et al., 2008] and lower bound [Calvanese et al., 2006, Ortiz et al., 2006b, Calvanese et al., 2007].

Regarding query matching w.r.t. capability advertisements in a $\mathcal{KB}$, one can give an account on the fact that there exists a match using the notion of intersection of sets as described in definition (4.1.3). Viewing the matching problem between desired capabilities formulated in a query and service capability advertisements stored in $\mathcal{KB}$ as determining intersection of sets is intuitive and helpful in understanding the problem of semantic service discovery, issues involved in matching and selection. Often it is even more interesting to give an additional account of the goodness of matches. As a result of research effort of [Paolucci et al., 2002, Grimm et al., 2004, Grimm & Hitzler, 2008], a standard matching model [Paolucci et al., 2002] has been proposed. This model consists of five types of matches representing different level of matching precision and compatibility. We introduce this model briefly for the purpose of refining the matching notion described in definition (4.1.3) where the concept of match is based on *compatibility* alone. With the standard matching model one can extend the matching semantics of intersection and account for more precise decision. We define A to denote the set of declared service advertisements and the set Q to denote the desired features within the query:

- *exact match*: if advertisement A and the query Q are equivalent sets, it is called an exact match and is denoted with

  $$A \equiv Q$$

  which denotes semantically that the specific capability formulated in the query set is exactly satisfied and covered by the capability declared in the advertisement set;

- *plug-in match*: if concept C in the query set Q is a subconcept of concept D in the advertisement set A, there is a plug-in match, i.e.,

  $$(\forall C \in Q \wedge \forall D \in A).C \sqsubseteq D$$

  . A plug-in match represents semantically the situation where advertisement is more general than the query set and that

  $$Q \subset A$$

  , i.e., the query set is totally contained in the advertisement set. The specific capability formulated in the query can be either partially or at best completely satisfied by the capability declared in the advertisement set;

- *subsume match*: if concept D in the advertisement set A is a subconcept of concept C in the query set Q, there is a subsume match with

  $$(\forall C \in Q \wedge \forall D \in A).C \sqsupseteq D.$$

  it represents semantically the fact that the query set is more general than the advertisement set and that

  $$A \subset Q$$

  , i.e., the advertisement set is just a subset of the query set with the significant meaning that there exists concepts in the query set which are not contained inside the advertisement set. The specific capability formulated in the query can be at best partially satisfied by the capability declared in the advertisement set;

- *intersection match*: if concept D in the advertisement set A can satisfy concept C in the query set Q, there is a intersection match with formally

  $$(\forall C \in Q \wedge \forall D \in A).\neg(C \sqcap D \sqsubseteq \bot).$$

  This match type corresponds semantically to the matching notion described in definition (4.1.3) representing that there exists an intersection between the two set

  $$Q \cap A \neq \{\emptyset\}$$

  while on the other hand conveying little information of high good the match is. The specific capability formulated in the query can be satisfied by the capability declared in the advertisement set, though there is no guarantee whether the requested capability will be entirely provided;

- *disjoint match*: it is characterized by complete disjointness between the two sets:

  $$Q \cap A = \{\emptyset\}$$

  which means a non-match and that the specific capability formulated in the query cannot be satisfied by the capability declared in the advertisement set.

We observe that exact match is the most desirable and preferable type of match which is followed by plug-in match and by subsume match and so forth such that the following relation exists

$$\textit{disjoint match} \prec \textit{intersection match} \prec \textit{subsume match} \prec \textit{plug-in match} \prec \textit{exact match}$$

with $\prec$ denoting the relation in terms of the degree of exactness and precision of matches.

4.2 Action Theoretic Foundations for Service Composition

If there is a client request of certain desired functionality which can not be satisfied by any single available service, it may still be possible that the functionalities of some of the available services, when combined together in an appropriate way, can satisfy the client's request. This is where *service composition* comes in. One tries to *synthesize* a composite functionality from the pieces of functionality of the available services in order to satisfy a more complex requirement on functionality. Formally we say that the available services can be *composed*, *coordinated* and *orchestrated* as a *virtual controller service* to serve the purpose of functionality synthesis. The essential question in a web service composition problem (WSC) is how to combine and services in which order so that basic constraints such as preconditions and postconditions of service invocation are observed. Without proper account for these constraints during design time and a strict observance of a suitable order during runtime, coordination of the component services is impossible. In order to shed light on the WSC problem, we harness a knowledge representation formalism [Levesque & Brachman, 2004] called the situation calculus [McCarthy, 1963, McCarthy, 2001a, McCarthy, 2001b, Reiter, 2001a] and the agent programming languages GOLOG/ConGolog [Levesque et al., 1997, Giacomo et al., 2000] to lay down a foundation framework for a formal knowledge relevant approach [Levesque & Lakemeyer, 2001] to this problem.

4.2.1 Situation Calculus

The formal language of the situation calculus $\mathcal{L}_{sitcalc}$ was proposed by John McCarthy in [McCarthy, 1963] which has gained appreciation in academia and is included in standard material in every introductory course on artificial intelligence such as [Russell & Norvig, 2002]. Since its inception it has evolved into a widely known and adopted formalism in the investigation of various technical problems in theoretical axiomatization about actions and their effects as well as in reasoning about dynamical aspects of systems. It has been taken seriously as a foundation for practical work in the domain of AI planning, system control, simulation, database theories, agent programming and robotics such as in [Reiter, 1996, Giacomo et al., 1997, Levesque et al., 1997, Lesperance et al., 1999, Sardina et al., 2004, Giacomo et al., 2009]. It is a based on a useful fragment of the first order logic language with some second order elements[6]. The situation calculus language $\mathcal{L}_{sitcalc}$ has been extended and thoroughly studied by the research community since McCarthy's first proposal, especially by the late Raymond Reiter who has made significant contribution to this branch of knowledge representation formalism in seminal worsks such as [Reiter, 2001a, Reiter & Pinto, 1993, Pinto & Reiter, 1995, Pirri & Reiter, 1999, Reiter, 1996, Reiter, 1998, Reiter, 2001b, Pirri & Reiter, 2000]. Since then it has been successfully used in many disciplines involving the specification of dynamical systems, especially in robotics, controller systems, simulation, etc., whereas our effective concerns are attributed to the works by McIlraith and her group in [McIlraith, 1999, McIlraith & Son, 2001, McIlraith & Son, 2002, McIlraith et al., 2001, Narayanan & McIlraith, 2002, Narayanan & McIlraith, 2003].

The basic language components of $\mathcal{L}_{sitcalc}$ are the following types:

- **action** for describing actions in $\mathcal{L}_{sitcalc}$, as a shorthand we generally use "a" to denote this type of language construct in the situation calculus;
- **situation** for situations in $\mathcal{L}_{sitcalc}$ with a shorthand "s" is used conventionally;
- **object** for everything else in the domain of interest.

[6] Noticeable is the alphabet of $\mathcal{L}_{sitcalc}$ includes countably infinitely many predicates variables of all *arities* For our purpose of service composition investigation, the second order elements are not of interest because it can incur undecidability [Levesque et al., 1998]. We concentrate on situation calculus predicates, otherwise called fluents will two variables which are proved to be decidable [Gu & Soutchanski, 2006].

The essential assertion mechanism is a *predicate* which is also common in traditional propositional logic. A predicate is evaluated according to its trueness, either being *true* or *false* and it consists of a countably finite number of places for variables with the total number called the *arity*. A predicate which value depends on $situation$ and varies is called a *fluent* in the situation calculus language $\mathcal{L}_{sitcalc}$. By convention the $situation$ argument is written in the place of the last argument in a fluent. There are a number of important foundational situation calculus predicates and functional symbols which are intrinsic to $\mathcal{L}_{sitcalc}$ and they are listed below:

1. a special binary function symbol "do" in $\mathcal{L}_{sitcalc}$ which is defined as $do : action \times situation \rightarrow situation$. With an arity of two, the definition domain of this function is $action$ and $situation$ in the situation calculus and the value domain is obtained from the cartesian product of them which is a $situation$. The interpretation of $do(a, s)$ lies in the evaluation of its side effect of the binary function "do" which semantically returns the successor situation resulting from performing action a in the situation s. An abbreviation for a 'zero' action is:

$$do([\,], s) \stackrel{def}{=} s$$

   and a shorthand form of writing a sequential execution of actions is:

$$do([a_1, a_2, \ldots, a_n], s) \stackrel{def}{=} do(a_n, do(\ldots, do(a_1, s)\ldots))$$

   where $do([a_1, a_2, \ldots, a_n], s)$ is called a *log*[7];

2. a special constant symbol "S_0" is used to denote the *initial situation*;

3. a binary predicate $\sqsubset$: $situation \times situation$ is evaluated to define an ordering relation on situations. The semantics of interpretation of this predicate is as action histories, for instance, $s \sqsubset s'$ means s is a proper subhistory of s';

4. a special binary predicate "$Poss$" in $\mathcal{L}_{sitcalc}$ which is defined as $Poss : action \times situation$. This predicate $Poss(a, s)$ is evaluated for trueness indicating whether is is possible to perform an action a in the situation s;

5. let n be a number for arity with $n \geq 1$, a $\mathcal{L}_{sitcalc}$ predicate $(action \cup object)^n$ with countably infinite arity that are *independent* of $situation$, for instance, $human(Reiter)$, $book(author(human(Reiter)))$, $title(languague(en), book(Knowledge\_in\_Action))\ldots$ indicates *relations* using situation calculus predicate;

6. let n be an arity with $n \geq 1$, the $\mathcal{L}_{sitcalc}$ function symbols called *action functions* with $(action \cup object)^n \rightarrow action$ asserts an $action$ in the value domain such as $pickup(x), move(loc1, loc2)$;

7. juxtaposing with the *action functions*, *situation independent function* specifies $(action \cup object)^n \rightarrow object$ and asserts $object$ in the value domain which is independent of situation calculus $situation$ such as $sqrt(x)$, $height(door)$, $depth(hole)$. *Situation independent function* is not interpreted in contrary to *action functions* as $action$ in $\mathcal{L}_{sitcalc}$;

8. a predicate with n arity where $n \geq 1$ which is defined as $(action \cup object)^n \times situation$. This type of predicates are called *relational fluents* or simply *fluents*[8] and are used to denote *situation dependent relations* such as $ontable(orange, s)$, $final(s)$ and $chairmanof(John, Corporate\_B, s)$. By convention the situation variable is placed in the position of the last argument in *fluents*;

[7] See([Reiter, 2001a], chapter 4 for detail.)

[8] In fact fluents belong to a class of important situation calculus mechanism which depend on a situation argument to be evaluated.

9. a function symbol with n arity where $n \geq 1$ which is defined as $(action \cup object)^n \times situation \rightarrow action \cup object$. This is called *functional fluents* and are used to denote *situation dependent functions* such as $sendToPrinter(printerNo, queueNo, taskNo, s)$.

The proper way to understand a situation in $\mathcal{L}_{sitcalc}$ is to view it as a *history*, namely, a sequence of actions. Two situations are identical iff they denote identical histories. Every situation in $\mathcal{L}_{sitcalc}$ corresponds to a sequence of actions, for instance:

$$do(pickup(x), do(drop(y), do(pickup(y), S_0)))$$

has the meaning starting from the initial situation S_0, the action $pickup(y)$ is performed which leads to a situation other than the initial situation. As mentioned previously this successor situation serves as the situation argument when the action $drop(y)$ is performed resulting again in a successor situation which is used in the outermost $do(a, s)$ function when the action $pickup(x)$ is performed last. Therefore the nested execution instructions of a history is both read and executed from the innermost to the outermost do function.

4.2.2 Basic Action Theories

A basic action theory is a set of situation calculus axioms which models the actions and their effects in a given dynamic system $\mathcal{D}$ together with *functional fluent consistency property*. Important types of situation calculus axioms are:

Definition 4.2.1 (Action Precondition Axiom). An action precondtion axiom $\mathcal{D}_{ap}$ of the situation language $\mathcal{L}_{sitcalc}$ is a formula of the form:

$$Poss(A(x_1, \ldots, x_n), s) \equiv \prod_A (x_1, \ldots, x_n, s)$$

where A is a situation calculus *action function* symbol with arity n representing with a situation calculus action function and $\prod_A(x_1, \ldots, x_n, s)$ a formula which is uniform in s[9] and whose free variables are among $x_1, \ldots, x_n, s$. The action function can be abbreviated by convention to:

$$A(x_1, \ldots, x_n), s) \Leftrightarrow A(\vec{x})$$

where $\vec{x} = x_1, x_2, \ldots, x_n$ denotes a shorthand to write the n-ary arguments and therefore the axiom can be rewritten to:

$$Poss(A(\vec{x}), s) \equiv \prod_A (\vec{x}, s)$$

The uniformity requirements on $\prod_A(x_1, \ldots, x_n, s)$ ensures that the preconditions for the executability of the action function $A(x_1, \ldots, x_n)$ are determined only by the *current* situation s. For each one primitive action $A(\vec{x})$ there is one action precondition axiom within the set $\mathcal{D}_{ap}$. World dynamics are specified by *effect axioms* which are closely related the *action precondition axioms* and describe the effects of a given action on the fluents which specify the causal laws of the domain. We can write an effect axiom in a general form:

$$Poss(A(\vec{x}), s) \wedge C(\vec{x}, s) \supset (\neg)F(\vec{x}, do(A(\vec{x}), s))$$

where $C(\vec{x}, s)$ is a first order situation calculus formula specifying the contextual conditions under which the action $A(\vec{x})$ will have its real *specified effect* on the trueness of fluent F.

[9] A first order situation calculus formula is uniform in s if it does not contain any term mentioning another term of type situation calculus $situation$ in itself.

□

Example 4.2.2. A simple example illustrating definition (4.2.1) is a blocks world which is shown here encoding an action precondition axiom:

$$Poss(pickup(x), s) \equiv (\forall x) \neg holding(x, s) \wedge \neg heavy(x, s)$$

Here the situation calculus *action function* $A(x_1, \ldots, x_n)$ is the action $pickup(x)$ and the axiom states that this action is possible iff the two situation calculus fluents on the right hand side hold, i.e., the agent is not currently holding x (encoded with the relational fluent $\neg holding(x, s)$ which must be true) and that x is not heavy (encoded with the fluent $\neg heavy(x, s)$). The axiomatization of action precondition in this form stems from the *qualification problem* [Russell & Norvig, 2002] in the artificial intelligence and its non-monotonic formulation. With the rationale of specifying world dynamics with effect axioms which have the general form:

$$Poss(A(\vec{x}), s) \wedge C(\vec{x}, s) \supset (\neg) F(\vec{x}, do(A(\vec{x}), s))$$

we show here some more examples of them:

$$\begin{array}{lcl} Poss(drop(p, x), s) \wedge fragile(x, s) & \supset & broken(x, do(drop(p, x), s)) \\ Poss(explode(b), s) \wedge nexto(b, x, s) & \supset & broken(x, do(explode(b), s)) \\ Poss(repair(p, t, x), s) \wedge toolsAvailable(t, s) & \supset & \neg broken(x, do(repair(p, t, x), s)) \end{array}$$

where the contextual conditions are $fragile(x, s)$, $nexto(b, x, s)$ and $toolsAvailable(t, s)$ respectively and they contribute to the fact that the specified effects on the right hand side of the fluents hold.

John McCarthy and Patrick Hayes first observed in [McCarthy & Hayes, 1969] that axiomatizing dynamical aspects of the world requires more than mere action precondition and effect axioms. Since the events of the world cannot be just characterized by action causal laws which remark changes to the state of the world, in a complete and sound characterization it is also necessary to state the *invariants* of the domain, i.e., the fluents which remain unaffected by a given action. *Frame axioms* are used to accomplish such characterization; without them it is not possible to specify which fluents and which states of the domain remain unchanged. This is called the frame problem [Scherl & Levesque, 1993, Scherl & Levesque, 2003, Russell & Norvig, 2002] in artificial intelligence. In order to have complete knowledge in an axiomatized knowledge base, it is not only necessary to provide account for causality of actions affecting specific fluents in a domain but also enumerate over those axioms which will not affect the fluents respectively.

Example 4.2.3. A typical example showing the frame problem in axiomatizing the action of a person dropping something that does not change the color of the thing:

$$Poss(drop(p, x), s) \wedge color(x, s) = c \supset color(x, do(drop(p, x), s)) = c$$

where the contextual condition here is a situation calculus functional fluent $color(x, s)$ which is evaluated to return the color denoted with c of x. The color c of x is not affected by the action of dropping x. Another example is to show dropping one thing will not affect another:

$$Poss(drop(p, x), s) \wedge \neg broken(y, s) \wedge [x \neq y \vee \neg fragile(y, s)] \quad \supset \quad \neg broken(y, do(drop(p, x), s))$$

this axiom states that one things x is dropped by a person p in the situation s and another thing y is not broken in that situation s and that the two things are different or y is not a fragile thing in the situation s then the action of dropping will not affect the thing y which remains not broken.

The problem of frame axioms is that there are a vast number of them. In fact only relatively few actions affect the trueness of a given situation calculus fluent while most of all other actions leave the trueness of the fluent unchanged. Domain axiomatization however must account for all these frame axioms; moreover theorem proving with a vast number of frame axioms makes it inefficient. There were historical suggestions of theoretical approach to solve the frame problem by [Haas, 1987, Schubert, 1990]. What counts as an acceptable solution to the frame problem is a systematic procedure for generating all the frame axioms from the situation calculus effect axioms. However, what is more important is that we try to obtain a *parsimonious representation* for the vast set of frame axioms. The late Raymond Reiter has proposed a simple and effective solution to the frame problem [Reiter, 1991] in the situation calculus in order to deal with the sheer number of frame axioms.

Definition 4.2.4 (Situation Calculus Solution to the Frame Problem). The effect axioms in the situation calculus can be written with an account on the possible contextual conditions which will positively and negatively affect the trueness of a fluent as in a general form:

$$\begin{aligned} Poss(A(\vec{x}), s) \wedge \gamma_F^+(\vec{x}, a, s) &\supset F(\vec{x}, do(a, s)) \\ Poss(A(\vec{x}), s) \wedge \gamma_F^-(\vec{x}, a, s) &\supset \neg F(\vec{x}, do(a, s)) \end{aligned}$$

where the contextual condition $\gamma_F^+(\vec{x}, a, s)$ is a first order situation calculus formula describing under what conditions that an action a in the situation s will affect the fluent F on the right hand side to become true in the successor situation $do(a, s)$. Analogously $\gamma_F^-(\vec{x}, a, s)$ represents the contextual condition under which the action a will affect the fluent F to take on a false value. Reiter's solution [Reiter, 1991] is based on the *completeness assumption* where the above characterizations in fact describe all the causal laws affecting the trueness of a specific fluent F.

□

Definition 4.2.5 (Successor State Axiom). Successor state axiom of the situation calculus language $\mathcal{L}_{sitcalc}$ is denoted with $\mathcal{D}_{ss}$. With the *completeness assumption* and the general parsimonious solution to the frame problem described in definition (4.2.4), successor state axiom can be described in the general form:

$$Poss(A(\vec{x}), s) \supset \left[F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \gamma_F^-(\vec{x}, a, s))\right]$$

where the formular $F(\vec{x}, do(a, s))$ can be abbreviated with the shorthand $\Phi_F(\vec{x}, a, s)$. There are mainly two types of successor state axioms:

1. A *relational fluent* F based successor state axiom for an (n+1)-ary formula has the form:

   $$F(x_1, \ldots, x_n, do(a, s)) \equiv \Phi_F(x_1, \ldots, x_n, a, s)$$

   where $\Phi_F(\vec{x}, a, s)$ is a situation calculus formula uniform in s, all of which free variables are among $a, s, x_1, \ldots, x_n$ (if arity is 0, the relational fluent F has one argument of the type $situation$ in $\mathcal{L}_{sitcalc}$. There is one successor state axiom for each relational fluent within the set $\mathcal{D}_{ss}$.

2. A *functional fluent* f based successor state axiom in the situation calculus language is an (n+1)-ary formula of the form:

   $$f(\vec{x}, do(a, s)) = y \equiv \Phi_f(\vec{x}, y, a, s)$$

   where the variable y denotes the effect, i.e., the return value of the functional fluent and $\Phi_f(\vec{x}, y, a, s)$ is a formula uniform in s, all of which free variables are among $\vec{x} = x_1, \ldots, x_n$ and y, a, s The value of a functional fluent in a successor situation is determined entirely by properties of the *current* situation s. There is one successor state axiom for each functional fluent within the set $\mathcal{D}_{ss}$.

□

Example 4.2.6. On the basis of definition of successor state axiom described in definition (4.2.5) this example shows a relational fluent and a functional fluent:

1. In an example about relational fluent we try to characterize the *effect* as successor state axiom for the action $a = drop(p, x)$ which characterizes the act of dropping by a person denoted with variable p and a thing denoted with variable x:

$$\begin{aligned} broken(x, do(a, s)) \quad \equiv \quad & (\exists p)\{a = drop(p, x) \wedge fragile(x, s)\} \vee \\ & (\exists b)\{a = explode(b) \wedge nexto(b, x, s)\} \vee \\ & \{broken(x, s) \wedge \neg(\exists p) a = repair(p, x)\} \end{aligned}$$

   This relational fluent states that a thing denoted with x will be broken as successor situation of $do(a, s)$ which can be an effect caused by either that x is fragile in situation s and the action taken is a person denoted with p dropping x or that there is a bomb denoted with b exploding and the bomb is next to the thing x in situation s or the thing x is already broken is the situation s and there is no person p who repairs x. The uniformity of the formula Φ_F guarantees the trueness of fluent $F(\vec{x}, do(a, s))$ in the successor situation $do(a, s)$ is determined entirely by the current situation s which is Markovian [Fadel & McIlraith, 2002].

2. An example illustrating the functional fluent is given:

$$\begin{aligned} height(x, do(a, s)) = y \quad \equiv \quad & a = moveToTable(x) \wedge y = 1 \vee \\ & (\exists z, h)(a = move(x, z) \wedge height(z, s) = h \wedge y = h + 1) \vee \\ & height(x, s) = y \wedge a \neq moveToTable(x) \wedge \neg(\exists z) a = move(x, z) \end{aligned}$$

   which can be interpreted as the successor situation of evaluating the height of a thing denoted with x by executing an action a in the situation s which return a value denoted with the variable y as an effect of the evaluation of the functional fluent $height(x, do(a, s))$. such an action can be the action $moveToTable(x)$ interpreted as moving x onto a table and y is evaluated to be 1 or there exists another thing z and a given height h for which the action of $move(x, z)$ interpreted as moving z onto x and evaluating the height of h will have the effect that y is equal to a new height $h+1$ or the height has already been evaluated to be y and no action is carried out, neither $moveToTable(x)$ nor $move(x, z)$.

Definition 4.2.7 (Unique Name Axioms). Unique name axioms in the situation calculus language $\mathcal{L}_{sitcalc}$ is denoted with $\mathcal{D}_{una}$. For any single n-ary action a, equivalence is defined as

$$a(x_1, \ldots, x_n) = a(y_1, \ldots, y_n) \supset x_1 = y_1 \wedge \cdots \wedge x_n = y_n$$

iff the arity for two argument lists is equal and all the elements in the argument lists are pairwise equal. For any distinct actions a and b and by using the abbreviated notation for an n-ary argument list $\vec{x} = x_1, \ldots, x_n$ and $\vec{y} = y_1, \ldots, y_n$ different names for both actions are defined as $a(\vec{x}) \neq b(\vec{y})$ which indicate that the arguments are pairwise disjoint and unequal. Since for any domain axiomatization there are predictably large number of such unique name axioms in order to guarantee that there is no naming conflicts, for parsimonious reason, it is conventionally assumed that these axioms are true without writing them out for every domain axiomatization.

□

Definition 4.2.8 (Axioms for Initial Situation). In the situation calculus language $\mathcal{L}_{sitcalc}$ axioms for initial situation is denoted with $\mathcal{D}_{S_0}$. They represent the set of first order formulas in which S_0 is the only term of type $situation$ and thus uniform in S_0. Aximos in the set $\mathcal{D}_{S_0}$ do not quantify over situations, or use the special binary fluents $do(a, s)$ or $Poss(a(\vec{x}), s)$. $\mathcal{D}_{S_0}$ can contain formulas which do not mention any situation term.

□

Definition 4.2.9 (**Basic Action Theory**). A basic action theory $\mathcal{D}$ in the situation calculus [Reiter, 2001a] is a set of first order axioms of the form:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una}$$

where

- Σ are the set of foundational axioms for the situation calculus;
- $\mathcal{D}_{S_0}$ is the set of situation calculus describing the initial situation as described in definition (4.2.8);
- $\mathcal{D}_{ap}$ is the set of action precondition axioms, one for each *action function* in $\mathcal{L}_{sitcalc}$ as described in definition (4.2.1);
- $\mathcal{D}_{ss}$ is the set of successor state axioms including the types *relation fluents* and *functional fluents*, with one for each fluent as described in definition (4.2.5);
- $\mathcal{D}_{una}$ is the set of unique name axioms for all *action function* with distinct action symbols, i.e., $A(\vec{x}) \neq B(\vec{x})$ where A and B are two distinct action symbols in the situation calculus language $\mathcal{L}_{sitcalc}$ as described in definition (4.2.7).

$\mathcal{D}$ satisfies the functional fluent consistency property of a functional fluent f which has the successor state axiom:

$$f(\vec{x}, do(a, s)) = y \equiv \Phi_f(\vec{x}, y, a, s)$$

where

$$\begin{aligned} \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \quad \models \quad & (\forall a, s).(\forall \vec{x}).(\exists y)\Phi_f(\vec{x}, y, a, s) \wedge \\ & [(\forall y, y').\Phi_f(\vec{x}, y, a, s) \wedge \Phi_f(\vec{x}, y', a, s) \supset y = y'] \end{aligned}$$

This consistency property offers a sufficient condition to rule out a source of inconsistency in the successor state axiom of the functional fluent f stating that the condition defining the fluent's value in the successor situation is unique.

□

4.2.3 Action Metatheory for the Situation Calculus

We first define satisfiability of a basic action theory in the situation calculus and with it we define an important operation called regression in the situation calculus.

Definition 4.2.10 (**Relative Satisfiability**). A basic action theory $\mathcal{D}$ in the situation calculus is relatively satisfiable iff $\mathcal{D}_{una} \cup \mathcal{D}_{S_0}$ is satisfiable.

□

An important theorem proving mechanism in the situation calculus is the *regression* operation. It provides a systematic way to establish that a basic action theory entails a regressable formula in the situation calculus language $\mathcal{L}_{sitcalc}$. *Regression* is also a central concept which forms the basis for many planning precedures [Manna & Waldinger, 1980, Pirri & Reiter, 2000] in artifical intelligence and for automated reasoning in the situation calculus [Pirri & Reiter, 1999]. Generally the regression of a situation calculus formula ϕ through an action a is an (a priori) formula ϕ' which holds prior to the action a has been performed iff the formula ϕ holds after executing a. The regression operation resembles the *backward reasoning* in *classical*

planning [Russell & Norvig, 2002] in artificial intelligence: given a goal state in which the reasoning process retraces the plan to figure out the initial state. The situation calculus successor state axiom supports regression operation in a natural way. Reiter [Reiter, 2001a] introduces a notation for the regression operation $\mathcal{R}$ and defines a regressable formula W in the situation calculus language $\mathcal{L}_{sitcalc}$ as follows:

Definition 4.2.11 (Regressable Formula in the Situation Calculus). According to [Reiter, 2001a], a formula W in the situation calculus language $\mathcal{L}_{sitcalc}$ is *regressable* iff

1. Every term of type $situation$ in $\mathcal{L}_{sitcalc}$ mentioned by W has the syntactic form:

   $$do([\alpha_1, \ldots, \alpha_n], S_0)$$

   where the arguments $\alpha_1, \ldots, \alpha_n$ is an n-ary list of type $action$ in the situation calculus;

2. For every predicate of the form $Poss(a, s)$ mentioned by W, α has the syntactic form $A(\vec{x})$ for an n-ary action function symbol A in $\mathcal{L}_{sitcalc}$;

3. W does not quantify over situations;

4. W does not mention the predicate symbol $\sqsubset$, nor does it mention any equation $\sigma = \sigma'$ where σ, σ' are of type $situation$ in $\mathcal{L}_{sitcalc}$.

□

The point in regressable formula is the fact that each of its $situation$ terms is rooted at the initial situation S_0 and therefore it can be inspected from the term exactly how many actions are involved. If a regressable formula mentions the predicate $Poss(a, s)$, it is possible to tell what the action symbol is by inspecting the first place in its arguments, for instance a regressable formula telling a history of someone walking from location A to B then enter Susan's office can be formulated as $Poss(enter(office(Susan)), do(walk(A, B), S_0))$.

Definition 4.2.12 (Regression Operator for Non-Functional Fluents in the Situation Calculus). Given a regressable formula W in the situation calculus language $\mathcal{L}_{sitcalc}$ and W does not contain functional fluent. The *regression operator* $\mathcal{R}$ when applied to W is determined relative to a basic theory of actions of $\mathcal{L}_{sitcalc}$ that serves a background axiomatization. Let $\vec{x}$ be a tuple of terms consisting of a list of arguments $x_1, \ldots, x_n$, α be a term of type $action$ and σ be a term of type $situation$. The regression operator for non-functional fluents is defined as:

1. if W is an atom, for W there are the possibilities:
   - W is a situation independent atom in terms of the type $action$ or $object$ whose predicate symbol is not a fluent, then

     $$\mathcal{R}[W] = W$$

   - W is a relational fluent depending on situation of the form $F(\vec{x}, S_0)$, then

     $$\mathcal{R}[W] = W$$

   - W is a regressable $Poss(A(\vec{x}), \sigma)$ predicate for the terms $A(\vec{x})$ of type $action$ and σ of type $situation$ respectively. A is an action function symbol in $\mathcal{L}_{sitcalc}$, then there is an action precondition axiom $Poss(A(\vec{x}), s) \equiv \prod_A(\vec{x}, s)$ in $\mathcal{D}_{ap}$, then the regression operator over W is

     $$\mathcal{R}[W] = \mathcal{R}\left[\prod_A(\vec{x}, \sigma)\right]$$

- W is a relational fluent of the form $F(\vec{x}, do(\alpha, \sigma))$ with the successor state axiom in $\mathcal{D}_{ss}$ written in the general form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, then the regression operator over W is

$$\mathcal{R}[W] = \mathcal{R}[\Phi_F(\vec{x}, \alpha, \sigma)]$$

2. if W is a non-atomic formula, regression is defined inductively.

$$\begin{array}{lcl} \mathcal{R}[\neg W] & = & \neg\mathcal{R}[W] \\ \mathcal{R}[W_1 \wedge W_2] & = & \mathcal{R}[W_1] \wedge \mathcal{R}[W_2] \\ \mathcal{R}[(\exists v)W] & = & (\exists v)\mathcal{R}[W] \end{array}$$

We observe that the regression operator applies only to regressable formulas, such a formula must only mention the initial situation S_0

□

Theorem 4.2.13. ***Uniformity of the Regression Operator*** *Given W is a regressable formula of $\mathcal{L}_{sitcalc}$ that mentions no functional fluents and $\mathcal{D}$ is a basic action theory. $\mathcal{R}[W]$ is a formula uniform in S_0 and*

$$\mathcal{D} \models W \equiv \mathcal{R}[W]$$

Proof. (sketch) According to [Reiter, 2001a] because W and $\mathcal{R}[W]$ are logically equivalent: the former is a regressable formula that is uniform in S_0 and the latter is the regression operator applied to W, for proving $\mathcal{D}$ it is sufficient to show that $\mathcal{R}[W]$ mentions only the term S_0 which is the case and most of the axioms in $\mathcal{D}$ would be irrelevant to the proof. □

Theorem 4.2.14. ***Regression Theorem*** *Given W is a regressable formula of $\mathcal{L}_{sitcalc}$ that mentions no functional fluents and $\mathcal{D}$ is a basic action theory, then*

$$\mathcal{D} \models W \;\; \textit{iff} \;\; \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \models \mathcal{R}[W]$$

Proof. (sketch) Based on the last proof to prove theorem (4.2.13) and the satisfiability of a basic action theory described in definition (4.2.10), it is obvious by substitution that theorem (4.2.14) is proved. □

Regression is very useful in the investigation of a basic action theory in order to prove that a sequence of primitive actions[10] is in fact executable and therefore represented as a history of executed actions.

Definition 4.2.15 (**Executable Ground Action Sequences in the Situation Calculus**)**.** Given an n-ary predicate $executable(\sigma)$ where σ is a variable of type $situation$ in $\mathcal{L}_{sitcalc}$, Σ the foundational axioms in $\mathcal{L}_{sitcalc}$ and n a natural number with $n \geq 0$, then

$$\Sigma \models (\forall a_1, \ldots, a_n).executable(do([a_1, \ldots, a_n], S_0)) \equiv \bigwedge_{i=1}^{n} Poss(a_i, do([a_1, \ldots, a_{i-1}]), S_0)$$

where $\vec{\alpha} = a_1, \ldots, a_n$ the sequence of *ground action* to execute, i.e., actions which do not contain arguments variables and let $\alpha_1, \ldots, \alpha_n$ be a sequence of ground actions in $\mathcal{L}_{sitcalc}$, then the sequence of ground action is *executable*

$$\mathcal{D} \models executable(do([\vec{\alpha}], S_0))$$

iff

$$\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \bigwedge_{i=1}^{n} \mathcal{R}[Poss(\alpha_i, do(\alpha_1, \ldots, \alpha_{i-1}, S_0))]$$

[10] The terminology of a primitive action is exchangeable with that of an atomic action throughout this thesis.

□

The above definition provides a systematic way for determining whether a sequence of ground actions and a ground situation S_0 is executable with $do([\alpha_1, \ldots, \alpha_n], S_0)$ based on regression operation. It reduces the test of executability to a theorem proving task by checking entailment of the union of the initial situation axioms and the unique name axioms. With a straighforward idea in the spirit of definition (4.2.12), we define the regression operator for functional fluents.

Definition 4.2.16 (Prime Functional Fluent Terms). A functional fluent is *prime* iff it has the form

$$f(\vec{x}, do([\alpha_1, \ldots, \alpha_n], S_0))$$

for $n \geq 1$ and each of the terms in $\vec{x}$ and $\alpha_1, \ldots, \alpha_n$ is uniform in S_0. Therefore prime functional fluents, S_0 is the only term of type $situation$ mentioned by $\vec{x}$ and $\alpha_1, \ldots, \alpha_n$.

□

Definition 4.2.17 (Regression Operator for Functional Fluents in the Situation Calculus). Given a regressable formula W in the situation calculus language $\mathcal{L}_{sitcalc}$ and W contains functional fluent, the regression operator is defined as:

1. if W is a regressable $Poss(A(\vec{x}), \sigma)$ predicate in $\mathcal{L}_{sitcalc}$ where A is an action function symbol and σ is a term of type $situation$ in $\mathcal{L}_{sitcalc}$, then regression operator is defined the same as for the respective sort of W in definition (4.2.12);

2. if W is a regressable predication beside a $Poss(A(\vec{x}), \sigma)$ predicate, there are three possibilities:
   - if S_0 is the only $situation$ term mentioned in W then
     $$\mathcal{R}[W] = W$$
     which is obvious based on the only $situation$ term S_0;
   - if W mentions a term of the form $g(\vec{\tau}, do(\alpha', \sigma'))$ for a functional fluent g and because W is regressable, $g(\vec{\tau}, do(\alpha', \sigma'))$ mentions a prime functional fluent term as described in definition (4.2.16). Let $g(\vec{\tau}, do(\alpha', \sigma'))$ have the form of a functional fluent $f(\vec{x}, do(\alpha', \sigma'))$ and let its successor state axiom be
     $$f(\vec{x}, do(\alpha', \sigma')) = y \equiv \phi_f(\vec{x}, y, a, s)$$
     , then the regression operator is defined as
     $$\mathcal{R}[W] = \mathcal{R}\left[(\exists y).\phi_f(\vec{x}, y, \alpha, \sigma) \wedge W|_y^{f(\vec{x}, do(\alpha, \sigma))}\right]$$ [11]
     It is necessary to focus on prime functional fluents to guarantee that the regression operation actually terminates;
   - if W is a relational fluent of the form $F(\vec{x}, do(\alpha, \sigma))$ and W does not mention any functional fluent of the form $g(\vec{\tau}, do(\alpha', \sigma'))$, then the regression operator is defined the same as in definition (4.2.12).

[11] In general given a situation calculus regressable formula ϕ, the notation $\phi|_t^{t'}$ represents the formula ϕ' that is obtained by replacing all t' with t in ϕ.

3. For non-atomic formulas, regression is defined inductively analogously as described in the last point of definition (4.2.12).

□

Another important mechanism of automatic deduction is the *projection problem* in artificial intelligence which is described in general as: given a sequence of ground action terms and a formula G, determine whether G is true in the situation resulting from performing the sequence of ground actions. In the situation calculus the *projection problem* is described in the following.

Definition 4.2.18 (Projection Problem in the Situation Calculus). Given a basic action theory $\mathcal{D}$ in $\mathcal{L}_{sitcalc}$ and let $\alpha_1, \ldots, \alpha_n$ be a sequence of ground action terms and $G(s)$ be a situation calculus formula with one free variable of type $situation$ in $\mathcal{L}_{sitcalc}$, the projection problem is to check the entailment of:

$$\mathcal{D} \models G(do([\alpha_1, \ldots, \alpha_n], S_0))$$

With the help of the definition of regressable formula described in definition (4.2.11), it $G(s)$ is a regressable formula, then the regression theorem in (4.2.14) provides an immediate mechanism to solve the *projection problem* in that it is necessary and sufficient to *regress* $G(do([\alpha_1, \ldots, \alpha_n], S_0))$ and check entailment of the regressed formula w.r.t. the union set of situation calculus initial situation axioms and unique name axioms:

$$\mathcal{D}_{una} \cup \mathcal{D}_{S_0} \models \mathcal{R}[G(do([\alpha_1, \ldots, \alpha_n], S_0))]$$

In artificial intelligence *planning*, verifying a proposed plan whether a sequence of actions $do([\alpha_1, \ldots, \alpha_n], S_0)$ satisfies a goal G is a natural application of projection. It is necessary and sufficient to show that the axioms of the planning domain entail the projection query $G(do([\alpha_1, \ldots, \alpha_n], S_0))$.

□

Example 4.2.19. Given a sequence of ground actions $\vec{\alpha}$ describing how to provide a person a cup of coffee and answer the query whether the person has become the cup of coffee after the actions are executed in the defined sequence, we can formulate in the situation calculus language $\mathcal{L}_{sitcalc}$:

$$\begin{aligned}\vec{\alpha} \quad = \quad & pourCoffee(blueMountainCoffee, mug); walk(Agent, office(Patrick)); \\ & enter(office(Patrick)); giveCoffee(Patrick, mug)\end{aligned}$$

where the notation ";" is used conventionally to denote an ordered sequence, we can query the basic action theory knowledge base with a projection query $hasCoffee(\vec{p}, s)$:

$$\mathcal{D} \models \mathcal{R}[hasCoffee(Patrick, do([\vec{\alpha}], S_0))]$$

If the regressed formula is entailed by the knowledge base $\mathcal{D}$, then the query returns true.

4.2.4 Action Dynamic Logic Language GOLOG

GOLOG is a high-level logic programming language based entirely on the action theory of the situation calculus for implementing complex behaviors for dynamical systems and domains. Programs in GOLOG decompose into primitives that in most cases refer to actions in the external world. These actions and their changes to the world are formulated together with an account of situations as precondition and successor state in axioms in first order logic so their effects can be formally reasoned about. This feature of GOLOG supports the specification of dynamic systems at the right level of abstraction. GOLOG programs are evaluated with a theorem prover. The programmer supplies a background basic action action theory axiomatizing

the application domain by stating precondition axioms, one for each action, successor state axioms, one for each fluent, axiomatized specification of the initial situation of the domain, unique name axioms indexsituation calculus!unique name axioms and a GOLOG main program specifying the agent behavior of executing the program. These required axioms are described in the definitions (4.2.1), (4.2.5) (4.2.8), (4.2.7) and (4.2.9) previously.

In the GOLOG language [Levesque et al., 1997], in order to capture the essence of algorithmic programs, syntactic contrusts with conditional testing, decision branching, loops and procedures have been added to GOLOG to mimic *complex actions* in the domain of interest. A *complex action* is any programmatic expression of the form of conditions:

$$\textsf{if } (predicate) \textsf{ then } \delta_1 \ldots \textsf{else } \delta_2 \ldots \textsf{endif}$$

or in form of loop:

$$\textsf{while } (predicate) \textsf{ do } \delta_1; \delta_2; \ldots \textsf{endwhile}$$

or in procedures with procedure name symbol and argument list:

$$\textsf{proc } procedureName(x) \ [\ \delta_1; \delta_2; \ldots\ ] \textsf{ endproc}$$

The notation δ_i is used to represent a *partial program* in GOLOG and ";" denotes sequence order of program execution. A complex action is represented intuitively with a ternary fluent

$$Do(\delta, s, s')$$

which has been defined as a *macro* to denote program execution parsimoniously. In general *macros* represent partial programmatic fragments which can be expanded into a formulas of the situation calculus. The fluent is quintessential in defining the execution of actions in a GOLOG program. The semantic of $Do(\delta, s, s')$ is as follows: the fluent holds whenever the situation s' is a terminating situation of complex action δ execution starting in situation s. Often complex actions can be non-deterministic because they can have several position executions terminating differently[12].
We define the necessary constructs of a GOLOG program and describe the meaning of each contruct in the following definition.

Definition 4.2.20 (GOLOG Program Syntactic Constructs and Semantics). The constituent parts of syntactic contructs of GOLOG are defined with their respective situation calculus semantics as follows:

1. *primitive action* is represented with "a":

$$Do(a, s, s') \stackrel{def}{=} Poss(a\,[s], s) \wedge s' = do(a\,[s], s)$$

   where the notation $a\,[s]$ denotes the result of restoring a situation argument s to any functional fluents mentioned by the action a[13];

2. *test action* is denoted with $\phi?$ to represent a condition to check:

$$Do(\phi?, s, s') \stackrel{def}{=} \phi\,[s] \wedge s = s'$$

[12] If one visualizes all possible sequences or combination of executions of a non-deterministic complex action δ, a tree of $situations$ in $\mathcal{L}_{sitcalc}$ can be built with different execution paths called execution trajectories which terminate in different situations.

[13] The situation argument restoration is accomplished by expanding the fluents in an axiom with a situation argument s, for instance, if $a = see(trafficLight(locationX))$ and $trafficLight$ is a functional fluent then $a\,[s]$ is $see(trafficLight(locationX, s))$.

where ϕ is an expression which stands for a suppressed formula stripped down of situation variables and $\phi[s]$ is a restored formula, for instance, if ϕ is

$$(\forall x).ontable(x) \wedge \neg on(x, A)$$

then $\phi[s]$ is restored to

$$(\forall x).ontable(x, s) \wedge \neg on(x, A, s)$$

where each fluent is expanded with a situation variable respectively;

3. *sequence* is an ordered execution of actions:

$$Do([\delta_1; \delta_2], s, s') \stackrel{def}{=} (\exists s^{ast}).(Do(\delta_1, s, s^{ast}) \wedge Do(\delta_2, s^{ast}, s'))$$

where δ_1 and δ_2 represent two partial GOLOG programs and s^* is a certain intermediate situation such that the execution of δ_1 starts in the situation s and terminates in the intermediate situation s^* and the second program δ_2 begins execution in the intermediate situation s^* and terminates in the situation s'. The order of execution of the sequence of programs is therefore determined by the conjunction on the right hand side;

4. *non-deterministic choice of two actions*:

$$Do((\delta_1|\delta_2), s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$$

where the choice of which program to execute first in decided non-deterministically;

5. *non-deterministic choice of action arguments*:

$$Do((\pi x)\delta(x), s, s') \stackrel{def}{=} (\exists x)Do(\delta(x), s, s')$$

where the non-deterministic operator π in $(\pi x)\delta(x)$ means an agent picks non-deterministically an individual x as argument for the program δ and execute $\delta(x)$;

6. *non-deterministic iteration*:

$$Do(\delta^*, s, s') \stackrel{def}{=} (\forall P).\{(\forall s_1)P(s_1, s_2) \wedge (\forall s_1, s_2, s_3)[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\} \supset P(s, s')$$

where the operator δ^* means execute the program δ zero or more times and the definition of execution semantic with transitive closure[14] means that executing the program δ zero or more times takes the situation from s to s';

7. *conditional of program flow* can be defined in terms of the previous syntactic contructs:

$$\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \text{ endif} \stackrel{def}{=} [\phi?; \delta_1] \,|\, [\neg\phi?; \delta_2]$$

where $\phi?$ is the test action defined previously and the semantic of the conditional is captured with the syntactic constructs on the right hand side stating that if the test action is true then program δ_1 is executed else if the test action fails δ_2 is executed;

[14] The transitive closure is a second order operator which is necessary because non-deterministic iteration is not sufficiently defined with first order construct [Levesque et al., 1997].

8. *while loop*:

$$\textsf{while } \phi \textsf{ do } \delta \textsf{ endwhile} \stackrel{def}{=} [[\phi?; \delta]^*; \neg\phi?]$$

where the syntactic constructs means that the test action $\phi?$ is checked; if it is true the program δ is executed and then the test action is checked again (as termination conditional for the while loop) and if it is still true, the iteration continues non-deterministically and the test action is again checked. If it is false, the while loop is terminated;

9. *procedures* in a GOLOG program:

$$\textsf{proc } P_1(\vec{v_1})\delta_1 \textsf{ endproc}; \ldots; \textsf{proc } P_n(\vec{v_n})\delta_n \textsf{ endproc}; \delta_0$$

where P_i denotes procedure names with $\vec{v_1}, \ldots, \vec{v_n}$ as formal parameters, $\delta_1, \ldots, \delta_n$ denotes a set of GOLOG program bodies and δ_0 denotes the main program body. We evaluate the program in the semantic of Do:

$$Do(\{\textsf{proc } P_1(\vec{v_1})\delta_1 \textsf{ endproc}; \ldots; \textsf{proc } P_n(\vec{v_n})\delta_n \textsf{ endproc}; \delta_0\}, s, s') \stackrel{def}{=}$$
$$(\forall P_1, \ldots, P_n).\, [\textstyle\bigwedge_{i=1}^{n}(\forall s_1, s_2, \vec{v_i}).Do(\delta_i, s_1, s_2) \supset Do(P_i(\vec{v_i}), s_1, s_2)] \supset Do(\delta_0, s, s')$$

where a GOLOG program consists of a main program body denoted with δ_0 and a set of procedures P_i with a vector of formal input parameters as arguments which are executed in an ordered sequence, each procedure itself consists of a partial program body δ_i respectively.

□

Example 4.2.21. Given a controller that serves an elevator, we show a set of procedures which define the actions of the controller:

- the program body $\delta = moveDown$ moves the elevator down one floor, we show a procedure $down$ which takes a parameter n representing the floor number to go down to:

$$\textsf{proc } down(n) \;\; (n = 0)?|down(n-1); moveDown \textsf{ endproc}$$

where the procedure recursively calls itself, decrementing n and execute the program body $\delta = moveDown$ to go down one floor, until the test action $\phi? = (n = 0)?$ is false.

- we show that the controller parks the elevator on the ground floor with the procedure $park$:

$$\textsf{proc } park(\pi m)\, [atFloor(m)?; down(m)] \textsf{ endproc}$$

here the procedure $park$ uses the previous procedure $down$ and we notice also that the non-deterministic choice of argument (πm) is used because the procedure does not dictate which parameter must be passed. Instead it allows an agent to choose non-deterministically what argument m to pass and checks whether the test action $atFloor(m)?$ is true. It executes the previous procedure $down(m)$ to move the elevator down $m \in \mathbb{N}$ floors. More examples can be found in [Levesque et al., 1997].

Definition 4.2.22 (Proper GOLOG Program). A GOLOG program δ is *proper* iff s and s' are the only free variables of type $situation$ in $\mathcal{L}_{sitcalc}$ mentioned in $Do(\delta, s, s')$.

□

Definition 4.2.23 (General Planning in Theorem Proving). Given a basic action theory $\mathcal{D}$ for a certain domain, a sequence of actions $\vec{a} \stackrel{def}{=} [a_1; \ldots; a_n]$ and a goal formula $\phi(s)$ with a single free $situation$ variable s, a *classical planning* task w.r.t. $\mathcal{D}$ is to evaluate and prove entailment of

$$\mathcal{D} \models Legal(\vec{a}, S_0) \wedge \phi(do(\vec{a}, S_0))$$
$$do(\vec{a}, S_0) \stackrel{def}{=} do(a_n, \ldots do(a_2, do(a_1, S_0)) \ldots)$$
$$Legal(\vec{a}, S_0) \stackrel{def}{=} Poss(a_1, S_0) \wedge \cdots \wedge Poss(a_n, do([a_1; a_2; \ldots; a_{n-1}], S_0))$$

therefore the general *planning task* resolves to finding a legal sequence of actions by *resolution* [Schöning, 2008, Russell & Norvig, 2002] which is

- *legal* in the sense that it is executable where each action is executed when its precondition (encoded with an action precondition axiom) is satisfied and
- the *goal formula* $\phi(do(\vec{a}, S_0))$ is satisfied and as a "side effect", it holds the final state (situation) resulting from performing the actions in the sequence.

□

Definition 4.2.24 (Evaluation and Termination of GOLOG Programs). Execution of a GOLOG program amounts to finding a *ground situation* term σ[15] such that

$$Axioms_{sitcalc} \models Do(program_{GOLOG}, S_0, \sigma)$$

where S_0 is the initial situation and σ a placeholder for a term of type $situation$ in $\mathcal{L}_{sitcalc}$. In general execution evaluation is accomplished by trying to prove the entailment [Reiter, 2001a]:

$$Axioms_{sitcalc} \models Do(program_{GOLOG}, S_0, s)$$

where σ is substituted with a concrete $situation$ term s denoting a certain terminating situation for the program. If a constructive proof is obtained, the *ground situation* term $s = do(a_n, \ldots do(a_2, do(a_1, S_0)) \ldots)$ is returned as a *binding* for the variable σ. It represents an ordered trace of actions from a_1 to a_n. Analogously on a concrete level, given a GOLOG program δ and a background basic action theory $\mathcal{D}$ of the situation calculus specifying a certain application domain as described in definition (4.2.9). If δ is a *proper* GOLOG program, its execution is evaluated relative to $\mathcal{D}$ by proving the entailment:

$$\mathcal{D} \models (\exists s) Do(\delta, S_0, s) \equiv (\exists s) Do(\delta, S_0, do(\vec{a}, S_0))$$

where starting from the initial situation S_0, execution of δ terminates in a certain situation s. It states that if $\mathcal{D}$ entails the complex action $Do(\delta, S_0, s)$, there exists an existentially quantified binding $s = do(\vec{a}, S_0) \stackrel{def}{=} do(a_n, \ldots do(a_2, do(a_1, S_0)) \ldots)$ containing the correct execution trace $\vec{a} = a_1; a_2; \ldots; a_n$ of δ.

□

By convention complex actions are expressed as program bodies δ in GOLOG which are a type of *macro* in order to maintain a brevity and a degree of simplicity which can be expanded into formulas of the situation calculus to express *complex behaviors*. A GOLOG program is interpreted with a theorem prover which is implemented on top of the logic programming language Prolog. Researchers of the Cognitive Robotics Group of the University of Toronto has implemented the GOLOG language. The GOLOG interpreter is shown in listing A.1 in appendix A.
Nevertheless, the described theorem proving task to find the correct execution trace for δ incurs much overhead for the GOLOG interpreter because the complex action $(\exists s) Do(\delta, S_0, s)$ stands for a quite complicated second order sentence in $\mathcal{L}_{sitcalc}$, it is desirable to reduce $\mathcal{D}$ in order to simplify the task. Fortunately, we can do this with Σ-elimination [Reiter, 2001a] where Σ is the set of foundational axioms for situations in $\mathcal{D}$.

[15] The ground situation term σ is therefore similar to the goal formula $phi(s)$ of planning described in definition (4.2.23) in that it specifies as a "side effect" the final situation resulting from executing the primitive actions in the sequence. Though in definition (4.2.23) there is no specific requirement or specification on the semantic of termination, here for the evaluation of GOLOG programs, it is necessary to bring in the termination criteria using the situation-transition semantics of $Do(\delta, S_0, \sigma)$.

Theorem 4.2.25. *Σ-**Elimination** Given a basic action theory $\mathcal{D} = \Sigma \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss}$ in $\mathcal{L}_{sitcalc}$ as described in definition* (4.2.9) *and δ is a proper GOLOG program, then Σ is eliminable:*

$$\mathcal{D} \models (\exists s) Do(\delta, S_0, s) \text{ iff } \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \models (\exists s) Do(\delta, S_0, s)$$

Proof. (sketch) A sketch to prove theorem (4.2.25) can be found in [Pirri & Reiter, 1999]. □

Definition 4.2.26 (GOLOG Program Syntactic Constructs). A GOLOG interpreter handles the following legal language constructs with action expressions e which corresponds to the previously described program body δ in the following description:

- $e_1 : e_2$ the symbol ":" corresponds to ";" in $Do([e_1; e_2], s, s')$ and denotes *sequential execution* of e_1 followed by e_2;
- $?(p)$ is the GOLOG language construct for *test action* where p corresponds to $Do(p?, s, s')$ and denotes a condition;
- e_1 # e_2 denotes *non-deterministic choice of actions* e_1 or e_2 and corresponds to $Do(e_1|e_2, s, s')$;
- $\mathsf{if}(p, e_1, e_2)$ is a GOLOG predicate construct which denotes *conditional execution* which corresponds to $Do([p?; e_1] \mid [\neg p?; e_2], s, s')$;
- $\mathsf{star}(e)$ denotes *non-deterministic iteration* which corresponds to $Do(e^*, s, s')$;
- $\mathsf{while}(p, e)$ denotes *while-loop iteration* which corresponds to $Do([[p?; e]^*; \neg p?], s, s')$;
- $\mathsf{pi}(v, e)$ denotes *non-deterministic choice of action argument* where v is a Prolog constant which acts as a GOLOG variable and e denotes a program body of primitive action. It corresponds to $Do((\pi v)e(v), s, s')$;
- a is the name of a user-declared *primitive action* used in a GOLOG *procedure*.

□

The GOLOG interpreter presupposes a background basic action theory $\mathcal{D}$ with available information content $\mathcal{D} = \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss}$ and evaluation amounts to theorem proving of a set of regressable formulas. Based on its Prolog implementation, the GOLOG interpreter exerts these assumptions on a GOLOG main program:

1. GOLOG programs axiomatizing an application domain in the situation calculus language $\mathcal{L}_{sitcalc}$ do not contain functional fluent and they contain only a finite number of relational fluents and action function symbols;
2. The basic action theory $\mathcal{D}$ has a closed initial database $\mathcal{D}_{S_0}$ meaning that $\mathcal{D}_{S_0}$ contains appropriate unique names axioms, it contains a definition for each non-fluent symbol and for each fluent F it contains $F(\vec{x}, S_0) \equiv \Psi_F(\vec{x}, S_0)$ where $\Psi_F(\vec{x}, S_0)$ is a first order formula that is uniform in S_0.

Definition 4.2.27 (GOLOG Main Program). A GOLOG main program δ_0 is expected to have the following parts [Levesque et al., 1997]:

1. a set of clauses declaring each primitive action using the Prolog construct $\mathsf{primitive\_action}(a)$ where a is a name of the primitive action;
2. a set of clauses specifying the initial situation as described in definition (4.2.8) and a set of unique names axiom clauses as described in definition (4.2.7);

3. a set of clauses of the form proc$(name, body)$ to define complex actions as procedures where $name$ is a name and $body$ the GOLOG code implementation for the procedure respectively;

4. a set of action precondition axioms corresponding to $Poss(a, s$ of $\mathcal{L}_{sitcalc}$ in definition (4.2.1), declared using the predefined predicate poss(a, S) each for every declared primitive action a with S as variable to range over all situations;

5. a set of successor state axioms corresponding to definition (4.2.5) for each fluent and situation, declared using the predefined predicate holds$(fluent, situation)$ with $fluent$ being any relational fluent and $situation$ a situation. The clause is written as:
   a) a set of clauses defining holds$(fluent, s0)$ where $s0$ is the predefined GOLOG construct of initial situation S_0, characterizing which fluents are true in the initial situation;
   b) a set of clauses in the form holds$(fluent, do(a, S))$ where a is further assigned a definition of a primitive action and S is a free situation variable. These clauses represents definitions for successor state axioms, one for each fluent.

□

4.2.5 Concurrency with ConGolog

ConGolog [Baier & Pinto, 1999, Giacomo et al., 2000, Lesperance et al., 2008] is an extended version of GOLOG language that incorporates a rich semantic of concurrency and interrupt handling capability. It has built in features to handle

- concurrent processes with possibly different priorities,
- high level interrupts,
- arbitrary exogenous actions.

The process concurrent model in ConGolog is implemented as interleavings of primitive actions in the component processes. A concurrent execution of two processes is one where the primitive actions in both processes occur, interleaved in the execution in a certain manner. Since the ConGolog language is based on GOLOG, it absorbs the GOLOG syntactic constructs as described in definition (4.2.20). Interleaved concurrent execution in ConGolog takes advantage of the *blocking* concept. While blocking of execution in GOLOG can happen, for instance, if a deterministic program δ executes and reaches a point where it is about to do a primitive action a in the situation s but $Poss(a, s)$ evaluates to false; or a test action $\phi?$ fails, the overall execution fails in GOLOG. In ConGolog the failed program δ is *blocked* and if another program δ' available and is ready to execute, the current interleaving execution can continue with δ'.

When concurrency is central to the interpretation of ConGolog programs, it is necessary to redefine a transition semantic for the execution of complex action $Do(\delta, s, s')$ defined in GOLOG. Such a transition semantic is based on defining single step of computation in contrast to directly defining computation of the entire execution. Regarding transition semantic there are two addition relational fluents to add to the *evaluation semantic* of GOLOG.

Definition 4.2.28 (ConGolog Transition Semantic). Given a program δ and a start situation s, transition semantic [Giacomo et al., 2000] uses two additional relational fluents to refine the evaluation semantic $Do(\delta, s, s')$ of GOLOG:

1. $Trans(\delta, s, \delta', s')$ which is associated with a given GOLOG program δ defines the relation of execution of a program δ, either a primitive action or test action, in the situation s so that it reaches the successor state s' and δ' which represents what remains of the program after δ is executed;

2. $Final(\delta, s)$ is a relational fluent which denotes termination of program δ in the final situation s.

The final situation is reached after a finite number of $Trans$ transitions from a starting situation coincide with those satisfying the $Do(\delta, s, s')$ relation. If a program does not terminate, then there is no final situation which satisfies that $Do(\delta, s, s')$ relation. The reflexive transitive closure of the transition semantic is denoted with $Trans^*$ where

$$Trans^* \stackrel{def}{=} \forall T.\left[\left[\top \supset T(\delta, s, \delta, s) \wedge Trans(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s')\right] \supset T(\delta, s, \delta', s')\right]$$

where $T()$ is a transition semantic operator and that $\top \supset T(\delta, s, \delta, s)$ is true[16] and through intermediate transitions with program δ'' in situation s'', it is possible to imply the transition semantic $T(\delta, s, \delta', s')$. Using $Trans^*$ and $Final$ the complex action $Do(\delta, s, s')$ can be rewritten as:

$$Do(\delta, s, s') \stackrel{def}{=} \exists \delta'.Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$$

The transition semantic therefore amounts to iteratively single-stepping through a program δ obtaining a remaining program δ' in the situation s' so that δ can legally terminate in situation s' (when $Final(\delta', s')$ is true).

□

For comprehensive reasons it is necessary to refine the GOLOG semantics described in definition (4.2.20) using the transition semantic [Giacomo et al., 1997]. An empty program "nil" is introduced to denote program termination[17]. We redefine the described program constructs using $Trans$ and $Final$ with detailed verbal explanations of these axioms given in [Giacomo et al., 2000]. For procedure-free programs transition semantic is shown for:

1. *empty program*:

$$Trans(nil, s, \delta', s') \equiv \bot$$

$$Final(nil, s) \equiv \top$$

2. *primitive actions*:

$$Trans(a, s, \delta', s') \equiv Poss(a\,[s], s) \wedge \delta' = nil \wedge s' = do(a\,[s], s)$$

$$Final(a, s) \equiv \bot$$

3. *test actions*:

$$Trans(\phi?, s, \delta', s') \equiv \phi\,[s] \wedge \delta' = nil \wedge s' = s$$

$$Final((\phi?, s) \equiv \bot$$

4. *sequence*:

$$Trans([\delta_1; \delta_2], s, \delta', s') \equiv \exists \gamma.\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')$$

$$Final([\delta_1; \delta_2], s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

[16] If no transition happens, i.e., the remaining program is the same as the original program and the situation remains the same, then $T(\delta, s, \delta, s)$ is always true.

[17] The symbol nil literally means that there is no more program that remains to be executed.

5. *non-deterministic branching*:

$$Trans([\delta_1|\delta_2], s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$$

$$Final([\delta_1|\delta_2], s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$

6. *non-deterministic choice of argument*:

$$Trans((\pi v).\delta, s, \delta', s') \equiv \exists x.Trans(\delta_x^v, s, \delta', s')^{18}$$

$$Final((\pi v).\delta, s) \equiv \exists x.Final(\delta_x^v, s)$$

7. *non-deterministic iteration*:

$$Trans(\delta^*, s, \delta', s') \equiv \exists \gamma.(\delta' = \gamma; \delta^*) \wedge Trans(\delta, s, \gamma, s')$$

$$Final(\delta^*, s) \equiv \top$$

Definition 4.2.29 (ConGolog Syntactic Constructs and Semantics). Given the constructs and semantics which are adopted from GOLOG described in definition (4.2.20), using the transition semantics described in definition (4.2.28), the additional constructs of ConGolog are shown with corresponding transition semantic using $Trans$ and $Final$:

1. *synchronized conditional*:

$$\textsf{if } \phi?_{sync} \textsf{ then } \delta_1 \textsf{ else } \delta_2$$

$$Trans(\left[\textsf{if } \phi?_{sync} \textsf{ then } \delta_1 \textsf{ else } \delta_2\right], s, \delta', s') \equiv \\ \phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s')$$

$$Final(\left[\textsf{if } \phi?_{sync} \textsf{ then } \delta_1 \textsf{ else } \delta_2\right], s) \equiv \\ \phi[s] \wedge Final(\delta_1, s) \vee \neg\phi[s] \wedge Final(\delta_2, s)$$

2. *synchronized loop*:

$$\textsf{while } \phi?_{sync} \textsf{ do } \delta$$

$$Trans(\left[\textsf{while } \phi?_{sync} \textsf{ do } \delta\right], s, \delta', s') \equiv \\ \exists \gamma.(\delta' = \gamma; \textsf{while } \phi?_{sync} \textsf{ do } \delta) \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s')$$

$$Final(\left[\textsf{while } \phi?_{sync} \textsf{ do } \delta\right], s) \equiv \\ \neg\phi[s] \vee Final(\delta, s)$$

   where $\phi?_{sync}$ is the synchronized version of the same conditional in GOLOG. Synchronization means that test actions in both the synchronized if-conditional and while-loop do not involve transition from one situation to another[19]. The evaluation of the test action condition and the first action of the branch chosen are executed in an atomic unit similar to "test-and-set" semantic in concurrent programming[20];

3. *concurrent execution of programs*:

$$(\delta_1 \;||\; \delta_2)$$

---

[18] The term δ_x^v means substituting v with x.

[19] We recapitulate that the general semantic of a test action $\phi?$ in GOLOG is defined as the transition: $Do(\phi?, s, s') \stackrel{def}{=} \phi[s] \wedge s = s'$. A test action progresses the situation from s to s' after the test has finished. Synchronized test actions in ConGolog are associated with the subsequent action of the program body in a test-and-set manner so that transition of situation happens only after the action in the branch chosen has been executed.

[20] Description of the transition semantic using $Trans$ and $Final$ for these axioms can be referenced in [Giacomo et al., 2000]

$$Trans([\delta_1 \ || \ \delta_2], s, \delta', s') \equiv$$
$$\exists\gamma.\delta' = (\gamma \ || \ \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee$$
$$\exists\gamma.\delta' = (\delta_1 \ || \ \gamma) \wedge Trans(\delta_2, s, \gamma, s')$$

$$Final([\delta_1 \ || \ \delta_2], s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

$$(\delta_1 \ \rangle\rangle \ \delta_2)$$

$$Trans([\delta_1 \ \rangle\rangle \ \delta_2], s, \delta', s') \equiv$$
$$\exists\gamma.\delta' = (\gamma \ \rangle\rangle \ \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee$$
$$\exists\gamma.\delta' = (\delta_1 \ \rangle\rangle \ \gamma) \wedge Trans(\delta_2, s, \gamma, s') \wedge \neg\exists\zeta, s''.Trans(\delta_1, s, \zeta, s'')$$

$$Final([\delta_1 \ \rangle\rangle \ \delta_2], s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

the construct $(\delta_1 \ || \ \delta_2)$ denotes concurrent interleaved execution of programs δ_1 and δ_2 while $(\delta_1 \ \rangle\rangle \ \delta_2)$ denotes prioritized concurrent execution with δ_1 having higher priority than δ_2. Priority restricts possible interleaving of two programs with the lower prioritized program executes only when higher prioritized program is either done or blocked;

4. *concurrent iteration*:

$$\delta^{||}$$
$$Trans(\delta^{||}, s, \delta', s') \equiv$$
$$\exists\gamma.\delta' = (\gamma \ || \ \delta^{||}) \wedge Trans(\delta, s, \gamma, s')$$

$$Final(\delta^{||}, s) \equiv \top$$

this construct resembles the non-deterministic iteration δ^* with the difference that here the instances of program δ execute concurrently rather than in sequence. In δ^* depending on the size of the sequence to execute, the iteration continues zero or more time using the complex action Do to enforce: $nil|\delta|(\delta;\delta)|(\delta;\delta;\delta)|\ldots$ with ";" indicating sequential execution. Concurrent iteration is characterized by: $nil|\delta|(\delta||\delta)|(\delta||\delta||\delta)|\ldots$ with the "||" symbol denoting concurrency;

5. *exogenous actions and interrupt*:

$$< \phi \rightarrow \delta >$$

where $< \phi \rightarrow \delta >$ is an interrupt consisting of two parts: ϕ is a trigger condition and a program body δ to execute once the trigger condition becomes true. Once it has completed it is ready to be triggered again. Should an interrupt never become true, it will not be triggered. One can think of interrupts in program as the sequence

$$\{start\_interrupts; (\delta \ \rangle\rangle \ stop\_interrupts)\}$$

with the described semantic of prioritized concurrency. However if an *exogenous event* occurs asynchronously without being part of a user specified program, one still can capture this semantic by defining a special program for exogenous events

$$\delta_{EXO} \stackrel{def}{=} ((\pi a).Exo(a)?; a)^*$$

. Then the user specified program can be executed interleaved concurrently as

$$\delta||\delta_{EXO}$$

. This is advantageous because an exogenous event often is not under control of δ at all. In this manner, exogenous event is well captured with the semantic of non-deterministic concurrency in ConGolog.

□

ConGolog interpreter can be implemented on top of Prolog, an interpreter based on the described transition semantic using $Trans$, $Final$ and Do is shown in listing B.1 in appendix B. The interpreter requires that the action preconditions axioms, successor state axioms, unique names axioms and initial axioms to be expressible in Prolog clauses.

Definition 4.2.30 (ConGolog Program Syntactic Constructs). A ConGolog interpreter based on transition semantic handles the following legal language constructs:

- nil denotes the empty program;
- act(a) denotes atomic action where a is an action argument;
- test(c) denotes a test action where c is a situation which is a fluent or one expression of the following: and(c_1, c_2), or$(c_1, c2)$, neg(c), all(v, c) or some(v, c) where v is a logical variable;
- seq(p_1, p_2) denotes sequence;
- choice(p_1, p_2) denotes non-deterministic choice of actions;
- pick(v, p) denotes non-deterministic choice of argument where v is a logical variable and p is a program term which uses v;
- iter(p) denotes non-deterministic iteration;
- if(c, p_1, p_2) denotes if-then-else conditional execution where p_1 represents the program to execute in the then-branch and p_2 represents the else-branch;
- while(c, p) denotes a while-loop;
- conc(p_1, p_2) denotes concurrency;
- prconc(p_1, p_2) denotes prioritized concurrency;
- iterconc(p) denotes iterated concurrency;
- pcall(p_args) denotes procedure call where p_args is a procedure name together with a possible list of arguments to invoke.

□

Definition 4.2.31 (ConGolog Application Program). A ConGolog program δ_0 is expected to have the following parts [Giacomo et al., 2000]:

1. a set of clauses describing the initial situation S_0 which can be atomic clauses or other complex clauses. The restriction by the intrinsic Prolog implementation on the initial database is that restriction of a close world assumption is forced on it;
2. a set of action precondition axioms specified with $Poss$ for every action a and situation s with one axiom for each primitive action using a variable to quantify over all situations;
3. a set of successor state axioms for each fluent where one clause is required for each fluent with variables for actions and situations;

4. a set of procedures in the form

$$\mathsf{proc}(p(X_1, \ldots, X_n), \delta)$$

where p is the procedure name, the n-ary list $X_1, \ldots, X_n$ contains formal parameters and δ the program of the procedure body.

□

4.2.6 Sequential Temporal Extension of Situation Calculus

Specification of time in knowledge systems is a challenging task because time does not stand still itself, this makes even specification of the real current time a problematic issue. There have been different attempts to specify time in knowledge representation, some of these can be found in [Hobbs & Pustejovsky, 2003, Hobbs & Pan, 2004, Pan & Hobbs, 2005, Hobbs & Pan, 2006, Pan et al., 2006, Pan, 2007].

In the situation calculus temporal reasoning amounts to the specification of time often related to process duration [Pinto & Reiter, 1993, Pinto, 1994]. A functional fluent $time(a, t)$ is introduced which denotes the time of occurrence of an action with $time(A(\vec{x}, t)) = t$ involving a particular action $A(\vec{x}, t)$ as an argument [Reiter, 1998]. This expression can be abbreviated as $time(a) \equiv time(a, now)$ where now is a special constant in $\mathcal{L}_{sitcalc}$ denoting the current instance time. it is convenient to define a function $start(s)$ denoting the start time of an action a such that

$$start(do(a, s)) = time(a)$$

here $start$ takes a $situation$ term and returns the current time.

In section 4.2.5 concurrency is described as interleaved giving an account of actions that are interleaved in a way that time is not accounted for specifically. There are many advantages to use interleaved concurrency, for instance, in complex actions some actions can serve as *enabling* conditions for another action to start such as cutting a string can cause the attached bottle start to fall, etc. An extension of the situation calculus to incorporate temporal reasoning makes a combination of instantaneous actions, explicit representation of time and an interleaved account of concurrency can contribute to a rich formal representation situation calculus language to describe *processes* that overlap in temporally complex manner which are very intrinsic to many real world dynamical systems.

Concurrency raises interesting issues in reasoning about multiple actions, one of the essential issues is whether the duration of two actions actually overlaps, or for instance that the time span of one action is entirely, partially or not at all contained in the time span of another action. The issue can get quite complicated if more actions are added to the problem. A representation device in the situation calculus to tackle and overcome this issue is by conceiving actions as *processes* that are represented by *relational fluents* in $\mathcal{L}_{sitcalc}$. The duration of a process is evaluated as the temporal displacement or time span between two *instantaneous* and *durationless* actions w.r.t. a relational fluent corresponding to the primitive action [Reiter, 2001a]. A $start$ action that initiates the process and a end action that terminates it. The initiation is evaluated as the $start$ action making the fluent of the primitive action to become true and the end actions renders it false, for instance, for the primitive action $walk(loc1, loc2)$ two durationless instantaneous actions $startWalk(loc1, loc2)$ and $endWalk(loc1, loc2)$ mark the initiation and termination of the primitive action and the relational fluent $walking(loc1, loc2, s)$ is associated with the primitive action. The action $startWalk$ turns the fluent $walking(loc1, loc2, s)$ true and $endWalk$ makes it false. With one drawback that one cannot exactly represent two overlapping actions if the $start$ and end actions coincide exactly with each other, though being an exception to this temporal extension approach, this situation is really not quite

often. Notice that even same duration of two actions does not imply that the $start$ and end of both actions exactly coincide with each other. With this rationale, a situation of the form

$$do([a_1, a_2, \ldots, a_{n-1}, a_n], S_0) \stackrel{def}{=} do(a_n, do(a_{n-1}, do(\ldots, do(a_2, do(a_1, S_0)) \ldots)))$$

can be understood as a *world history* of actions execution starting in S_0 and after a non-deterministic period of time, a_1 occurs when it is started with it $start$ action and then it terminates when its end action makes the relational fluent false; subsequently after a non-deterministic period of time a_2 is initiated and then terminated in the same manner and so forth.

The above process concept of action initiation and termination w.r.t. to the duration that actions elapse is a solid foundation to temporal reasoning, however it is silent on an exact representation of the time at which actions occur. A function symbol $time(a) = t$ is introduced to represent an instantaneous action returning the time t when an action a occurs. Temporal modification to the $Do(a, s, s')$ macro in $\mathcal{L}_{sitcalc}$ to accomodate time can be defined as:

$$Do(a, s, s') \stackrel{def}{=} Poss(a, s) \wedge start(s) \preceq time(a) \wedge (s' = do(a, s))$$

where Do is extended in a temporal dimension with the functions $start(s)$ and $time(a)$ to a sequential situation calculus where:

$$start(do(a, s)) = time(a)$$

The symbol "$\preceq$" denotes an *order between situations* giving a partial ordered relation between, for instance, $s \preceq s'$ meaning that one can get to s' by a sequence of possible actions starting in situation s. The $start(s)$ function is analogous to a shorthand $now(s)$ which returns the current time t. A GOLOG program clause can be rewritten, for instance, w.r.t. the modified Do complex action from:

```
do(E,S,do(E,s)) :-
    primitive_action(E), poss(E,S) to
do(E,S,do(E,s)) :-
    primitive_action(E), poss(E,S), start(S,T1), time(E,T2), T1 <= T2
```

Augmenting ConGolog with a temporal dimension is beneficial for the upcoming discussion on service composition techniques using GOLOG and ConGolog. Not that this sequential temporal approach is indispensable but just it can enrich the semantic of a language to make it suitable for more realisitc modeling of dynamical systems such as web services.

4.2.7 Service Composition in the Situation Calculus

We can turn our attention to use the situation calculus as axiomatizing theory for the domain of services and use GOLOG/ConGolog to tackle web services composition problem. Researchers of Sheila Ann McIlraith's group have first proposed using an extended GOLOG[21] to tackle the web service composition problem by conceiving it as AI planning task [McIlraith et al., 2001, McIlraith & Son, 2002, Baier & McIlraith, 2006a] using the theoretical foundation of the situation calculus and *agent theories* [Wooldridge & Jennings, 1994, Wooldridge & Jennings, 1995, Lesperance et al., 1999, Fagin et al., 2003]. In their synthesis approach, web services are modeled as generic procedures of GOLOG programs with an appropriate axiomatized background basic action theory [Reiter, 2001a] such that their preconditions, postconditions, successor states and world altering effects are suitably axiomatized. They introduced an extended semantic in GOLOG to describe transition semantic in order to incorporate user constraints, sensing actions by planning agent, notion

[21] For brevity reason the term GOLOG henceforth means also ConGolog when it is mentioned in the following text.

of knowledge self-sufficient and physically sufficient programs.

In an *rational agent* approach to tackle the web service composition problem, we resort to a type of *model-based programming* in which a *model-based program* is a reusable hight-level program that captures the procedural knowledge of how to accomplish a task without the requirements of complete and detailed specification of it. Model-based programs are instantiated in the context of a *model* of a specific system and state of the world where the program instance is *sequences of actions* which can be performed by an agent to control the *behavior* of the system. A model-based program in contrary to conventional algorithmic programs is not necessarily *deterministic* and it comprises:

- a **model** is an abstract representation of the domain system being programmed, the operator or control actions that affect it and the state of the system. A model M of a model-based program is a situation calculus domain axiomatization in the language $\mathcal{L}_{sitcalc}$;
- a **program** is a collection of high level procedural algorithms for performing tasks using operators and control actions that are intrinsic to the model. A GOLOG program is a manifestation of such a program that is based on the model M of the situation calculus action theory $\mathcal{D}$.

It is obvious that using model-based program, we can tackle the problem of web service composition.

Web services are programs offering their functionalities accessible to clients over networks, especially the internet. They offer their operations as functionalities that are exposed as *behavior* to an agent, in this regard an intelligent *rational agent* can use an instance of the program, i.e., composition of web services realized in sequences of actions, as a plan to control the behavior of the service. What the plan is satisfying is essentially the goal of the composition which is the overall desired functionality of the behavior of a system and its state of the world. The problem of web service composition is often characterized by incomplete initial knowledge both in terms of functionality of available services as well as the constraints about using these services. Therefore web service composition must account for the ability to sense in the planning process by agents since web services also sense and accumulate knowledge to react upon input in the world besides just performing their world altering actions. McIlraith has shown how to represent and compile services into generic GOLOG programs in order to treat services as planning operators to allow agents to reason upon generation of possible execution plan. Input and output parameters of services are modeled as *generic procedures* [McIlraith & Son, 2002] with input as knowledge preconditions and output as knowledge effects in the planning context respectively. Exogenous events such as failure, timeouts, etc. can affect the things in the world being sensed and hence affect value of fluents. User input and user constraints are essential in pruning the search space of plans.

Definition 4.2.32 (GOLOG Tree Program). A GOLOG *tree program* is a program that does not contain any *unbounded loop* contructs [Baier & McIlraith, 2006a]. A tree program is guaranteed to terminate within a bounded number of iterations over a sequence of situations which can be unfolded into a well-formed situation tree[22] after performing a sequence of actions in a situation history. The *boundedness* of iteration is defined as

$$\mathsf{if}\ \phi? \wedge k \neq 0\ \mathsf{then}\ \{\mathsf{while}_k\ [\phi? \wedge k > 0]\ \mathsf{do}\ \delta; k-1\ \mathsf{endwhile}\}\ \mathsf{else}\ nil\ \mathsf{endif}$$

where $k \in \mathbb{N}$.

□

[22] See [Reiter, 2001a] chapter 4 for a definition.

Definition 4.2.33 (Deterministic GOLOG Tree Program). A *deterministic* GOLOG *tree program* is a program that does not contain any *non-deterministic choice of arguments* construct, i.e., the $(\pi x)\delta(x)$ contruct.

□

For a given complex action δ that terminates in $s' = do_{ca}(\delta, s)$ starting from s, we can analyze the possibility of execution of the complex action δ as:

$$Do(\delta, s, do_{ca}(\delta, s)) \vee (\neg\exists s''.Do(\delta, s, s'') \wedge \neg executable(do_{ca}(\delta, s)))$$

where $do_{ca}(\delta, s)$ is a named version for the predefined do action function in $\mathcal{L}_{sitcalc}$ denoting a complex action δ is performed in situation s, juxtaposing with the $do(a, s)$ for primitive actions. The above axiom states a epistemic *complete* account of the execution of a complex action δ, i.e., either the δ is executable within the Do computational semantics starting from situation s and reaches the successor situation $do_{ca}(\delta, s)$ which is implicitly s' as stated previously; or δ is not executable, i.e., there exists no successor situation s'' such that within Do the execution of δ starting from s cannot reach situation s'' and therefore $do_{ca}(\delta, s)$ is not executable. The predicate $executable(s)$ denotes a situation with all of its actions in the situation history s^*, i.e., all the a-priori situations in the situation tree [Reiter, 2001a] that lead to the current situation argument s are possible:

$$executable(s) \stackrel{def}{=} (\forall a, s^*).Poss(a, s^*) \wedge do(a, s^*) = s^{a-priori} \preceq s$$

where the symbol "$\preceq$" denotes the ordering relation for situations $s^{a-priori}$ and s within the situation tree where $s^{a-priori}$ is a certain a-priori situation within the situation tree relative to the current situation s. It follows that this account of do_{ca} computational semantic in $\mathcal{D}_{ca}$ can be added to the set of axioms of the basic action theory $\mathcal{D}$. Henceforth we call it the *physical executability* of a complex action within $\mathcal{D}_{ca}$.

Definition 4.2.34 (Physical Executability of Complex Actions). A given complex action δ is physically executable in all situation s iff

$$\mathcal{D} \models \forall s.executable(s) \wedge Do(\delta, s, do_{ca}(\delta, s)) \rightarrow executable(do_{ca}(\delta, s))$$

where $\mathcal{D}_{ca}$ is already merged within $\mathcal{D}$ and $do_{ca}(\delta, s)$ is defined previously.

□

Customizable Programs with User Constraints

To capture user constraints, [McIlraith & Son, 2002] extends $\mathcal{L}_{sitcalc}$ with a new fluent $Desirable(a, s)$ one for each action to specify that an action a is desirable in situation s. It is used to describe that a is not only physically executable in s but also desirable so that it helps to constraint search space for actions when realizing a GOLOG program. The fluent is generally true, i.e., $Desirable(a, s) \equiv true$ unless otherwise specified. The set of $Desirable$ fluents is contained in $\mathcal{D}_D \subseteq \mathcal{D}$. User constraints can therefore be expressed in two parts:

1. a set of *necessary conditions* ω in $\mathcal{D}_{necessary_D}$ of necessary conditions for desirable actions with $Desirable(a, s) \supset \omega_i$ where subscript i is an index into $\mathcal{D}_{necessary_D}$, for instance, a necessary condition to desire buying a new CD is that it is not yet available in my collection:

$$Desirable(buyCD(disc), s) \supset \neg existInCollection(disc, s);$$

2. a set of *personal constraint* $\Omega_{personal}$ in $\mathcal{L}_{sitcalc}$ specifying personal preferences as constraint expressions $C(s)$ with $C(do(A(\vec{x}), s))$. For instance, desired to buy if cheaper than 20 dollars and having money: $enoughMoney(s) \wedge costLessThan(disc, 20, s)$.

Definition 4.2.35 (Desirable Actions). Given a set of user constraints w.r.t. a basic action theory $\mathcal{D}$ containing the constraints $\mathcal{D}_{necessary_D}$ and $\mathcal{D}_{personal_D}$, an action that is $Desirable(a, s)$ is defined as

$$Desirable(a, s) \equiv (\bigvee_{i=1}^{n} \omega_i) \wedge (\bigwedge_{C(s) \in \mathcal{D}_{personal}} \Omega_{personal})$$

where $\Omega_{personal} = \mathcal{R}\left[C(do(A(\vec{x}), s))\right]$, i.e., a regression of the personal constraint expression using successor state axioms.

□

With the previous definition, the transition semantic for GOLOG, i.e., $Trans(\delta, s, \delta', s')$ and $Final(\delta, s)$, can incorporate the concept of desired actions:

$$\begin{aligned} Trans(a, s, \delta', s') &\equiv Poss(a, s) \wedge \delta' = nil \wedge s' = do(a, s) \\ Trans(a, s, \delta', s') &\equiv Poss(a, s) \wedge Desirable(a, s) \wedge \delta' = nil \wedge s' = do(a, s) \end{aligned}$$

so that the expression

$$Legal(a, s) \equiv Poss(a, s) \wedge Desirable(a, s)$$

can be used as a shorthand whereas for $Final$ it really does not matter whether it is desirable because it is about checking whether the program terminates rather than if we want it to terminate.
They introduce a new *order connective* which is denoted by ":" to loosen up the overly constrained sequential execution of primitive actions imposed by using ";" in GOLOG. The rationale is that it can somehow continue execution without letting the whole GOLOG program to fail after a certain individual action in the sequence has failed. If the first primitive action in the sequence fails, it deals with the failure by inducing the GOLOG program to search for sequence of actions replacing the failed action in order to enable the second action to execute. This feature can positively impact the GOLOG program for service composition because services do fail in practice quite often and having this order construct can induce the GOLOG interpreter to search in its search space for alternative known action trajectories; this can be understood in practice to mimic some *fail over handling*. The semantic for conventional sequential execution is $a_1; a_2$ iff $Poss(a_2, do(a_1, s))$ is true. However in conventional GOLOG, if it is not, a_2 can never be executed. Instead the GOLOG interpreter could search for an alternative sequence of actions $\vec{a} = [a'_1, \ldots, a'_n]$ to *achieve* the precondition such that a_2 can execute, i.e., $Poss(a_2, do(\vec{a}, do(a_1, s))) = true$. The *order connective* ":" for actions a_1 and a_2 can be written out to:

$$a_1; \textsf{while } (\neg Poss(a_2, s)) \textsf{ do } (\pi a') \left[Poss(a', s)?; a'\right] \textsf{ endwhile}; a_2$$

This search functionality to compensate failure can be encoded into a function

$$achieve(\phi) \stackrel{def}{=} \textsf{while } (\neg\phi?) \textsf{ do } (\pi a') \left[Poss(a', s)?; a'\right] \textsf{ endwhile}$$

with one drawback: this type of undirected search [Russell & Norvig, 2002] is that the search space can impose some overhead to the GOLOG interpreter.
Consequently the transition semantic of GOLOG can be further refined w.r.t. the new order connective and defined as:

$$Trans(\delta : a, s, \delta', s') \equiv Trans((\delta; achieve(Poss(a, s))); a, s, \delta', s')$$

$$Final(\delta : a, s) \equiv Final(\delta; achieve(Poss(a, s)); a, s)$$

Subsequently $achieve(Poss(a, s))$ can be replace by

$$achieve(Legal(a, s))$$

to incorporate the user constraints described previously.

Agent Sensing Actions in Service Composition

Web service composition problem in the situation calculus is often characterized with insufficient initial knowledge of a planning agent or when exogenous events exist that unpredictably change portion of the world state known to the agent or affect values of fluents that are known to the agent. Therefore web service composition is characterized by an interleaved sequence of *knowledge gathering actions* and *world-altering actual actions*, i.e., the agent performs beside the complex actions that represent the invocation of a service, it also performs actions in parallel to gather information about possible unpredictable state changes in the world. In this regard [McIlraith & Son, 2002] has proposed the idea of *self-sufficient* GOLOG programs to make it generic enough so that nothing is assumed about what the agents know in terms of their acquired knowledge to execute the programs. Preconditions for actions to be executed by a program is realized within the program as *kernel initial state* $Init_\delta$ w.r.t. the program denoting the preconditions for executing δ.
The knowledge of an agent regarding trueness of a formula ϕ and its belief state can be expressed with a distinguished fluent [Scherl & Levesque, 1993, Lesperance et al., 1999]

$$KWhether(\phi, s).$$

It characterizes completely the epistemic state of an agent since

$$KWhether(\phi, s) \stackrel{def}{=} Know(\phi, s) \vee Know(\neg\phi, s)$$

where $Know(\phi, s)$ is a special fluent representing the agent's knowledge w.r.t. the formula ϕ in the situation s, i.e., that the agent knows and believes that ϕ is either true or false.

Program *self-sufficiency* is characterized with the predicate

$$ssf(\delta, s)$$

and defined semantically over the structure of δ as follows:

$$\begin{array}{lcl}
ssf(nil, s) & \equiv & \top \\
ssf(\phi?, s) & \equiv & KWhether(\phi, s) \\
ssf(a, s) & \equiv & KWhether(Legal(a, s)) \\
ssf([\delta_1; \delta_2], s) & \equiv & ssf(\delta_1, s) \wedge \forall s'.\, [Do(\delta_1, s, s') \supset ssf(\delta_2, s')] \\
ssf([\delta_1 : \delta_2], s) & \equiv & ssf(\delta_1, s) \wedge \forall s'.\, [Do(achieve(Poss(\delta_1, s)), s, s') \supset ssf(\delta_2, s')] \\
ssf(\delta_1 | \delta_2, s) & \equiv & ssf(\delta_1, s) \wedge ssf(\delta_2, s) \\
ssf(\delta^*, s) & \equiv & ssf(\delta, s) \wedge \forall s'.\, [Do(\delta, s, s') \supset ssf(\delta^*, s')] \\
ssf((\pi x)\delta(x), s) & \equiv & \forall x.ssf(\delta(x), s) \\
ssf(\textsf{if}\ \phi\ \textsf{then}\ \delta_1\ \textsf{else}\ \delta_2\ \textsf{endif}, s) & \equiv & KWhether(\phi, s) \wedge (\phi(s) \supset ssf(\delta_1, s)) \wedge (\neg\phi(s) \supset ssf(\delta_2, s)) \\
ssf(\textsf{while}\ \phi\ \textsf{do}\ \delta\ \textsf{endwhile}, s) & \equiv & KWhether(\phi, s) \wedge (\phi(s) \supset ssf(\delta, s)) \wedge \\
 & & (\forall s'.\, [Do(\delta, s, s') \supset ssf(\textsf{while}\ \phi\ \textsf{do}\ \delta\ \textsf{endwhile}, s')])
\end{array}$$

Before executing a program an agent's belief state and level of knowledge is characterized by the part of knowledge which it has already possessed and another which it still needs to acquire [Lesperance et al., 1999]. In the former case, the source of information is obviously the state of the world prior to execution of any action. The agent acquires this information from the *kernel initial state* $Init_\delta(S_0)$ in the initial situation S_0. In the latter case, a customizable and generic GOLOG program does not assume the level of knowledge or the *belief state* of an agent regarding execution of the program. Instead it presupposes that an agent will take necessary steps to acquire additional information via sensing subsequently [Baier & McIlraith, 2006a] if it needs to do so.

Definition 4.2.36 (Knowledge Self-Sufficient GOLOG Programs). A GOLOG program δ is knowledge self-sufficient (KSSF) w.r.t. to a basic action theory $\mathcal{D}$ in $\mathcal{L}_{sitcalc}$ and kernel initial state $Init_\delta$ iff δ is a *deterministic tree program* as described in definition (4.2.33) and

$$\mathcal{D} \models Init_\delta(S_0) \wedge ssf(\delta, S_0)$$

This property ensures that given proper $Init_\delta$ the execution of a GOLOG program will not fail due to lack of knowledge of an agent, though a program will still fail due to impossible action, i.e., for instance $\neg Poss(\delta, s)$. It shows the epistemic knowledge of an agent knowing whether $Poss$ is true.

□

Definition 4.2.37 (Physically Self-Sufficient GOLOG Programs). A GOLOG program δ is physically self-sufficient (PSSF) w.r.t. to a basic action theory $\mathcal{D}$ in $\mathcal{L}_{sitcalc}$ and kernel initial state $Init_\delta$ iff

$$KSSF(\delta, Init_\delta) \wedge \mathcal{D} \models \exists s'.Do(\delta, S_0, s')$$

such that $PSSF(\delta, Init_\delta) \supset KSSF(\delta, Init_\delta)$.

□

Definition 4.2.38 (Self-Sufficient GOLOG Programs). Given a GOLOG program δ, a corresponding basic action theory $\mathcal{D}$ and a *kernel initial state* $Init_\delta$, the program is *self-sufficient* w.r.t. $\mathcal{D}$ and $Init_\delta$ iff

1. δ is knowledge self-sufficient as described in definition (4.2.36);
2. δ is physically self-sufficient and physically executable as described in definitions (4.2.37) and (4.2.34) respectively.

□

To execute the extended GOLOG program there is a choice between online or offline interpreter. Reiter suggests to use an online GOLOG interpreter to reason with sensing actions [Reiter, 2001a]. Comparing to an offline interpreter the online one does not usually support backtracking while offline interpreter suffers from computational overhead due to large search space and the inaptitude to automatically incorporate sensing actions. McIlraith proposes to use a combination of online and offline interpreter, coining the term *middle ground* (MG) which is a type of hybrid interpreter. It executes sensing actions for knowledge gathering for agents online while the actual actions are performed to simulate their world-altering effects[23]. The outcome of the online simulation is a complete execution plan where sequence of world-altering actions can be subsequently executed by real invocation of the corresponding services. The complete plan reveals a sequence of complex actions that are related to a sequence of existing web services. An agent *knows* how to execute the plan in order to achieve its goal through acquisition of information at runtime [Lesperance et al., 2000]. A MG GOLOG interpreter achieves a possible composition by continuously executing sensing actions and actual complex actions to simulate and search for a set of actions (services) which satisfy a goal w.r.t. a basic action theory and user constraints. If $do(\vec{\alpha}, S_0)$ is the termination situation after executing a GOLOG program δ with basic action theory $\mathcal{D}$ using the hybrid MG GOLOG interpreter starting in the initial situation S_0, then the entailment holds

$$\mathcal{D} \models \exists s.Do(\delta, S_0, do(\vec{\alpha}, S_0)) \wedge s = do(\vec{\alpha}, S_0) \wedge Legal(\vec{\alpha}, S_0) \wedge ssf(\delta, S_0)$$

where the sequence of actions $\vec{\alpha}$ comprises both sensing and world-altering actions in an interleaved manner, for instance $[s_1, a_1, a_2, a_3, s_2, \dots]$ where s_i denotes sensing action and a_i actual action respectively. We

[23] By world-altering action, it is meant here that a web service is represented as an action which is executed to have a perceivable effect in the changed state of the world (domain), whereby service is this respect is modeled quasi as a black box.

denote the sequence of sensing actions with $\vec{\alpha}_{sensing}$ and the sequence of world-altering actions with $\vec{\alpha}_{wa}$ where the outcome sequence $\vec{\alpha} = \vec{\alpha}_{sensing} \cup \vec{\alpha}_{wa}$. Sequence $\vec{\alpha}$ exists in output if this entailment relation holds

$$MG(\mathcal{D}, \delta, Init_\delta, \vec{\alpha}) \stackrel{def}{=} \mathcal{D} \models Init_\delta(S_0) \wedge Do(\delta, S_0, do(\vec{\alpha}, S_0))$$

If no unpredictable exogenous events or sensor error change the value of the fluents, then for all fluents F in $\mathcal{D}$:

$$\mathcal{D} \models F(\vec{x}, do(\vec{\alpha}_{wa}, \vec{\alpha}_{sensing}, S_0))$$

For the hybrid MG GOLOG interpreter, amendments have been made to the interpreter that have been incorporated into listing B.1 in appendix B whereby explanations are found in [McIlraith & Son, 2002].

The model M of our GOLOG program is based on an axiomatized background basic action theory $\mathcal{D}$ and we are interested in obtaining an *instance* of service composition program as a *controller* for solving the web service composition problem (WSC) so that an agent can use it as a plan to simulate and execute actions.

Definition 4.2.39 (Model-Based GOLOG Program Instance for WSC). Let δ be a model-based GOLOG progam w.r.t. the model M based on the basic action theory $\mathcal{D}$ in $\mathcal{L}_{sitcalc}$. An execution trajectory $\vec{A} = [a_1, \ldots, a_n] \cup [s_1, \ldots, s_n]$ is model-based program instance representing a solution to a web service composition problem (WSC) iff

$$M \models Do(\delta, S_0, do([a_1, \ldots, a_n], S_0))$$

Generation of such an instance $\vec{A}$ from the model M is the theorem proving task to prove:

$$\mathcal{D} \models \exists s'.Do(\delta, S_0, do(\vec{A}, S_0)) \wedge s' = do(\vec{A}, S_0) \wedge Legal(\vec{A}, S_0) \wedge ssf(\delta, S_0)$$

The extrated sequence $\vec{A}$ from the binding for s' is a sequence of actions (including actual and sensing actions) among the *trajectories* of a *situation tree* over all possible situations. The sequence of actions represents a *controller* of a compsite service which can be used by an agent to control the behavior of it. In addition, the extracted sequence can be verified that it will not pass through unsafe states, i.e., those with failure potential, by using a first-order logic formula $P(s)$ to include a situation s and proving:

$$M \models (\forall s').Do(\delta, S_0, s') \wedge s' = do(\vec{A}, S_0) \supset P(s')$$

□

The following propositions are derived from definition (4.2.39) regarding the existence of a model-based program instance and the aspect of goal achievement of a composite service program.

Proposition 4.2.40 (Existence of Program Instance). *A program instance exists for a model-based program δ and model M iff*

$$M \models (\exists s).Do(\delta, S_0, s)$$

□

Proposition 4.2.41 (Goal Achievement of Program). *Let $G(s)$ be a first-order logic formula representing the goal state of a model-based program δ with model M. The program and model are guaranteed to achieve the goal iff*

$$M \models (\forall s).Do(\delta, S_0, s) \supset G(s)$$

□

Proposition 4.2.42 (Goal Achievement of Program Instance). *Let $G(s)$ be a first-order logic formula representing the goal state of a model-based program δ with model M. A model-based program instance $\vec{A} = [a_1, \ldots, a_n] \cup [s_1, \ldots, s_n]$ of the program δ achieve the goal $G(s)$ iff*

$$M \models Do(\delta, S_0, do(\vec{A}, S_0)) \supset G(do(\vec{A}, S_0))$$

□

4.3 The Roman Model

After a survey of an agent based view of service composition in the action theory of the situation calculus, we turn our attention to another important approach which essentially let us formulate a client specification of a target service representing the desired functionality that we want to synthesize from available service by using deterministic finite state machines techniques to specify and synthesize a composition. In their effort to determine a sound, tractable and decidable automated service synthesis formalism for process oriented services, the group of researchers around Giuseppe De Giacomo at the Università di Roma La Sapienza has proposed a model of automated service synthesis based on description of conversational behavior of stateful services, i.e., by taking care of the constraints on a sequence of operations imposed by a set of available *stateful* service. This model is known as the *Roman Model* [Calvanese et al., 2008b] in academia.

The Roman Model is based on the concept of coordinating and orchestrating the *atomic interactions* between service client and a set of available services which form a *service community*. The central focus of this model is on the *delegation* of execution of actions, to the services in the service community which can perform the required operations. The *conversational behavior* is represented by finite state machine (FSM) with a finite set of transitions. Each transition in the finite state transition system represents a possible operation between the client and an available service. A community of services consists of a set of such finite state transitions systems describing the corresponding service. These services share a common set of action alphabets. In order to realize a composite service, the client specifies a virtual *target service* consisting of the client's desired *deterministic* operations which is represented as a finite state transition system that shares the common operation alphabets with the service community. The goal of composition synthesis is therefore to maintain the client's interaction with the virtual target service by using an *orchestrator* to *delegate* actions to the available services.

4.3.1 Characterizing Services

Services are characterized regarding their dynamic *conversational behavior* in the *Roman Model*. A service is a fragment of process oriented application logic intended to interact with a client agent atomically where a client is either a human user or another service.

Definition 4.3.1 (**Service**)**.** Let E be the notation of a service. A service is defined as a software artifact delivered and accessible over a network that interacts with its client agent C in order to perform a specified task by executing an *atomic actions* a. A client agent can be either a human user or another service E'. The finite set of all *atomic actions*[24] offered by a service E is called its *action alphabet*, denoted with Σ.

□

An interaction between client agent and a service can characterized as follows:

1. in a certain step of execution, an available service possesses an internal state; it proposes a finite set of executable operations among those in its current state to a client agent,
2. the client agent chooses one of these operations and requests the service to perform it,
3. the service executes the chosen operation on behalf of the client agent,
4. the service proceeds to a new state that is conformant to its behavioral specification, i.e. the set of its allowed states according to the specifiation of its finite state transition system,

[24] i.e. all the allowed executable operations of E

5. the service reiterates the suggestion to the client agent of the executable operations for the client to choose,

6. the client can again choose an operation or (in some specified state called the final state according to the specification) it can choose to end the interaction with the service.

Definition 4.3.2 (**Abstract Service Behavior**). Let E be a service, the *abstract service behavior* of E is represented as *deterministic transition system* (TS) where the service is observed as a black-box. Atomic actions performed by the service is modeled as state transitions, each having a corresponding enabling state as precondition and a successor state as effect of the transition caused by execution of the action. Service behavior is characterized with the tuple $(\Sigma, S_E, s_E^0, \delta_E, F_E)$ where:

- Σ denotes a finite set of operations of the service called the set of atomic actions which is contained in the *action alphabet* defining the TS;
- S_E denotes a finite set of states of E;
- s_E^0 denotes the initial state of E;
- δ_E with $\delta_E \colon S_E \times \Sigma \rightarrow S_E$ denotes a partial transition function of TS, when given a state s and an atomic action $a \in \Sigma$ returns the successor state s' resulting from the execution of a in state s, i.e., formally $\delta_E(s, a) = s'$;
- $F_E \subseteq S_E$ denotes the finite set of final states of E where the interactions of a client with E can be legally terminated.

□

From a conceptual perspective, the characterization of enterprise services in the domain of service oriented computing determines the following manifestations:

- *Service schema* specifies the feature of an enterprise service regarding the functional and non-functional requirements. The *Roman Model* deals with the functional requirements solely. It defines interface and behavior specification of an enterprise service.
- *Service implementation* and *deployment* represents the realization of an enterprise services in terms of software applications corresponding to the service schema as well as the technology platform used to deploy the service.
- *Service instance* represent a run time concept as an *object* of the implemented enterprise service with a unique identity. An instance realizes the implemented behavior as specified in the service schema. A deployed enterprise service can have multiple instances during runtime.

Definition 4.3.3 (**Service Community**). Let E_i be a service with the *action alphabet* Σ_{E_i}. A *service community* $C = \{E_1, \ldots, E_n\}$ is the finite set of available services which:

- shares a common set of *community action alphabet* Σ_C, with $\Sigma_C = \Sigma_{E_1} \cup \cdots \cup \Sigma_{E_n}$;
- identifies a specific service with E_i whose *alphabet* is conformed to the *alphabet* of C, i.e., $\Sigma_{E_i} \subset \Sigma_C$;
- a service E_i exports it *service behavior* in terms of Σ_C when joining the community.

□

Definition 4.3.4 (**Service Schema**). Let E be a service and Σ_E be its *action alphabet*. The service schema denoted with $\mathcal{A}^E$ specifies the functional requirements of a service E. $\mathcal{A}^E$ has two facets: from client's perspective it represents E's exported *service behavior* denoting the sequence of executable *atomic actions* $a \in \Sigma$ with constraint on their invocation order and from a service's perspective, it represents the *implementation* and *realization* of the service, specifying whether a is performed by E itself or whether it delegates a for execution by another service.

□

It is relevant to specify whether each atomic action is execute by a service E itself or whether its execution is delegated to another service E' in the community C with which E interacts as client agent transparently on behalf of its own client agent. From an external perspective of a client agent, a service E in C is a black box with a certain exported behavior; from an internal perspective due to implementation and activation of service instances to handle client request, it is necessary to describe how the atomic actions which are part of the the exported behavior are actually executed. To capture these two perspectives, A service schema $\mathcal{A}^E$ is divided into *external schema* and *internal schema*. A service instance inherits from the these schemas an external and internal view respectively when activated.

Definition 4.3.5 (**External Schema**). Let E be a service and Σ_E be its *action alphabet*. An *external schema* of E, denoted with $\mathcal{A}^{E_{ext}}$, is a formal specification of the behavior of service E in terms of all the atomic actions underlying Σ_E provided by E and the constraints of the invocation order of these atomic actions.

□

Definition 4.3.6 (**Internal Schema**). Let E be a service, Σ_E be its *action alphabet* and C be a service community to which E belongs. An *internal schema* of E, denoted with $\mathcal{A}^{E_{int}}$, is a formal specification of the internal implementation of the exported behavior of service E taking into account which atomic action is executed by E itself and which can be delegated to other services in C for execution.

□

In order to visualize the execution of the atomic actions of a service, i.e. the operations and its corresponding transitions of state within the service instance better, the notion of a *labeled execution tree* is used to show the behavior of a service in terms of the actions executable at each step and its transitions of internal states. A labeled execution tree is shown in figure 4.1. It is possible to visualize all possible sequences of deterministic atomic actions of a service by unfolding the deterministic finite state machine (FSM) that is associated with the service into an execution tree of sequences of actions. Each node in the tree represents a state corresponding to a state in the FSM. An (partial) incoming edges leading to the node representing an executed action that leads to the specified state and an outgoing edge from the node indicates a possible action in the current state to execute next. Each execution path of the unfolded FSM is call a *trajectory* of the execution tree. It constitutes a sequence of such edges visualizing the execution order of atomic actions and the corresponding state transitions.

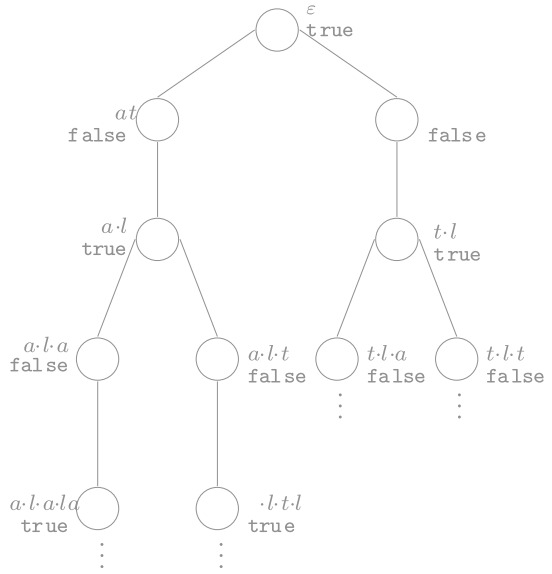

Figure 4.1: A labeled execution tree

Definition 4.3.7 (**Labeled Execution Tree**). Let E be a service and Σ_E be its *action alphabet*. An *execution tree*, denoted with $\mathcal{T}$ over the alphabet Σ_E , represents the *trajectory* of action execution paths of the service in interaction with its client. The characteristics of $\mathcal{T}$ are:

- the root node of $\mathcal{T}$ is denoted by ε and represents the fact that no action has been executed by the service;
- each node possesses an internal state $s_{node} \in \{true, false\}$ denoting whether the node is final, i.e., whether the client can stop terminate the interaction with the service if $s_{node} = true$, otherwise not;
- each edge of $\mathcal{T}$ is labeled with an *atomic action* $a \in \Sigma_E$ meaning that the execution of a is possible from the state of the incoming node; reaching the successor state in the successor node after executing a;
- each node x of $\mathcal{T}$ beside ε is labeled by a *labeling function* f which assigns a history of actions executed so far by the service up to x. For instance the history of a node x is denoted with $a \cdot l \cdot a$ where $a, l, a \in \Sigma_E$ are atomic actions and the history means that starting from ε the trajectory is followed by first executing the action a, then l and then a again up to node x;
- the number of trajectories and therefore the number of nodes of $\mathcal{T}$ can be infinite if the client continues interaction with E without choosing to terminate the interaction;
- a *labeled execution* tree is a pair $(\mathcal{T}, f)$ where every node of $\mathcal{T}$ is assigned a history by f and every edge is labeled properly with allowed actions.

□

Definition 4.3.8 (**External Execution Tree**). Let $\mathcal{A}^{E_{ext}}$ be the *external schema* of service E with *action alphabet* Σ_E. An *external execution tree* $T^{ext}(E)$[25] is a *labeled execution tree* defined over Σ_E as described in definition (4.3.7) obtained by unfolding the *deterministic finite state machine* (FSM) of E. It is represented with a pair $(\mathcal{T}, f)$ formally where $\mathcal{T}$ is the execution tree and f the *labeling function* assigning to each edge a possible action.

□

The *external execution tree* specifies the information about the execution of atomic actions w.r.t. states and state transitions of a service from an abstract perspective where each atomic action represents an executable operation of the service, abstractly grouped in the *action alphabet*, i.e., he set of all known and executable operations of a service. An external execution tree is shown in figure 4.2 where the nodes represents states and the edges with alphabet letters represent abstract named executable actions. Final nodes denote states where interaction with the service is allowed to terminate and are indicated with double circular nodes in the tree. In this abstraction the service itself is viewed as a kind of black box in the sense that all the perceivable characteristics of the service and the effects which the service has upon the state of the world are essentially captured within the possible interaction of the service with a client. The *external*

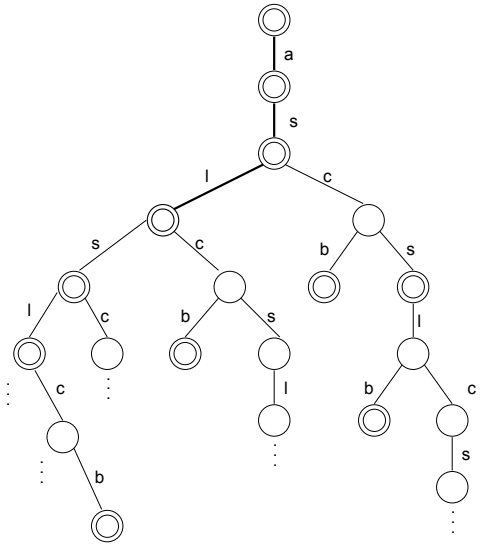

Figure 4.2: An external execution tree

[25] the abbreviated form is T^{ext} and it is constructed by $T(\mathcal{A}^{E_{ext}})$

execution tree encodes the information on the execution of possible actions at each internal state of the service which is opaque to the client. The client solicits the service to perform an action by sending an operation request to the service; whereby the only perceivable outcome of the execution to the client is often a message reply. This information is not observed by the external execution tree since messages constitute a series of dataflow between client and service and possibly within the service itself. What is more interesting is the dynamic aspect of the interaction between service and client. Since many process oriented services undergo transition of states within the process without notification to the client, it is important to encode in information of what service observable state permits which action to execute. This information is exactly encoded in the *external execution tree*. It is helpful in specifying constraints such as preconditions and effects of the service operations to allow more elaborate combination of different services with regard to these constraints.

Definition 4.3.9 (**Internal Execution Tree**). Let $\mathcal{A}^{E_{int}}$ be the *internal schema* of service E with *action alphabet* Σ_E. An *internal execution tree* $T^{int}(E)$[26] is a *labeled execution tree* defined over Σ_E analogous to definition (4.3.7). Additionally each edge is labeled by a pair (a, I) where a is the executable atomic action as in the case of an *external execution tree*. I is a nonempty set with $I = (E_i, e_i)$ denoting the services identified with their subscripts and their corresponding instances to execute a. Each service instance is identified uniquely with an instance identifier w.r.t. to service E_i to which the instance belongs.

□

The *internal execution tree* specifies the information about which service instances in a service community can execute each given action. The specification of which instance to delegate the execution of a certain action to is practical since a service is sometimes not able to perform the requested action itself either due to state constraints or unavailability of such a action. An example internal execution tree is shown in figure 4.3.

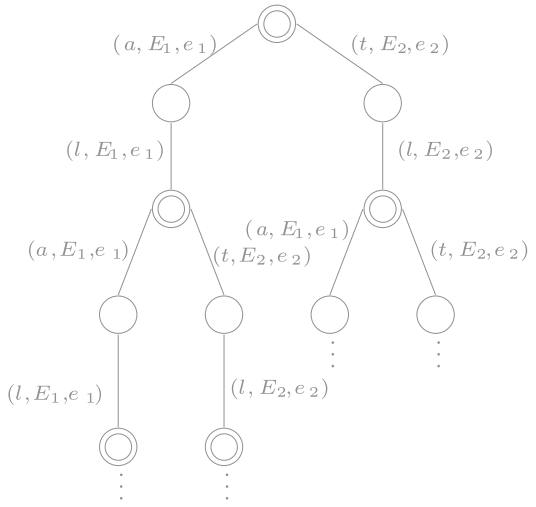

Figure 4.3: An internal execution tree

The service delegable instances are identified with their corresponding instance identifier. The set I in definition (4.3.9) is nonempty due to the fact that there can be more than one service capable of delegating a requested action to. If a service executes the action itself, then the identifier is this for the service instance. The notion of an internal execution tree T^{int} *induces* an external execution tree T^{ext} can be understood as the external execution tree obtained from T^{int} by removing the part labeling the service instances from T^{int}, keeping only the information on actions. Such induced external execution tree is called an *offered external execution tree*.

Theorem 4.3.10 (**Service Conformance**). *Given an internal execution tree T^{int} and an external execution tree T^{ext} of a service E defined over the common alphabet Σ_E and T^{int} is inducible, T^{int} conforms to T^{ext} iff T^{ext} is topologically equal to the offered external execution tree of T^{int} obtained by induction.*

Proof. (sketch) Suppose T^{int} is inducible, the *offered external execution tree* of T^{int} is obtained by dropping the information on service instances contained in T^{int}. Since T^{int}, similar to T^{ext}, is a *labeled execution tree* which is defined over the set of alphabet Σ_E as defined in definition (4.3.7), after removing all instance information, T^{int} is topologically transformed into its corresponding T^{ext}; thus T^{ext} is equal to the *offered external execution tree*. □

Definition 4.3.11 (**Service Well-Formedness**).

[26] the abbreviated form is T^{int}

Let E be a service and T^{int} be the *internal execution tree* and T^{ext} be the *external execution tree* of E, a service is *well-formed* if its T^{int} *conforms* to its T^{ext}.

□

Lemma 4.3.12 (**Projection Operation**). *Given T^{int} of a service E and an execution path p starting from root ε in T^{int}, the projection operation on instance e_i of service E_i of an execution path is defined as the path p obtained by removing each edge of T^{int} on the path whose labeled pair (a, I) with the set I not containing instance e_i. After every removal, the start and end node of the removed edge is collapsed.*

Proof. (sketch) Straightforward from topological transformation using projection of the labeled execution tree defined in definition (4.3.7). □

Theorem 4.3.13 (**Service Coherency**). *The internal execution tree $T^{int}(E)$ of a service E is coherent with a service community C if:*

- *for each labeled edge (a, I), the action a is defined in the community alphabet Σ_C and for each pair within set I, $E_i \in C$*
- *for each execution path p in $T^{int}(E)$ from the root node ε to a node x on p, and for each pair $I = (E_i, e_i)$ appearing in labeled edges on p that are not this[27], the projection of p on instance e_i is a path in the corresponding external execution tree $T^{ext}(E_i)$ of service E_i from root to a node y in $T^{ext}(E_i)$. Additionally, if node x is final in $T^{int}(E)$, then node y in $T^{ext}(E_i)$ is final.*

Proof. (sketch) Straightforward from topological transformation using projection of the labeled execution tree. It can be shown that the projection operation returns a path in $T^{int}(E)$ that corresponds to a path in $T^{ext}(E_i)$. In fact the path in $T^{ext}(E_i)$ shows that execution can be delegated to service E_i. □

Theorem 4.3.14 (**Service Delegation**). *A service E in a service community C correctly delegates execution of actions to other services in C if the internal execution tree $T^{int}(E)$ of E is coherent with C.*

Proof. (sketch) Straightforward from the last proof. □

Definition 4.3.15 (**Service Community Well-Formedness**). A service community C is *well-formed* if each service in C is *well-formed* and the internal execution tree of each service is *coherent* with C.

□

4.3.2 Formalization of Service Interaction Dynamics

A service instance represents the dynamical aspect of a service running an interacting with its client. A running instance corresponds to an execution tree a highlighted current node within the execution tree, representing the point reached by execution up to a moment.

Definition 4.3.16 (**Service Instance**). A service instance is the tuple $e_i = (e_{i_{id}}, View_{ext}, View_{int}, node_{cur})$ and is characterized by :

- $e_{i_{id}}$ is an instance identifier;
- $View_{ext}$ denotes the *external view* of the instance which is represented as an external execution tree with a current node;

[27] i.e. the identifier for the instance itself

- $View_{int}$ denotes the *internal view* of the instance which is represented as an internal execution tree with a current node;
- $node_{cur}$ is the current node denoting the state of the service within the execution tree.

□

The life cycle of a service is characterized by:

1. activation the service instance,
2. offer of choice of the evocable actions,
3. termination of the service instance.

While the steps (1) and (3) of the life cycle is performed once, step (2) can be repeatedly performed. In the following these life cycle steps are described cursorily as a conceptual interaction protocol. Figure 4.4 shows this life cycle.

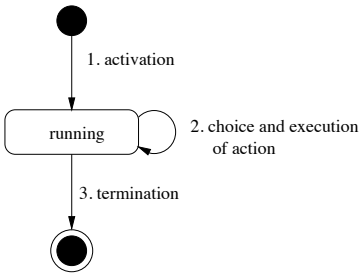

Figure 4.4: Life cycle of a service instance

Activation: It is required to create the service instance.
The client invokes activates the instance by sending a command:

$$\textbf{activate}\ \ E_i$$

A new instance e_j of service E_i is created and necessary resources for the execution of e_j are allocated. Each created service instance creates a copy of internal and external execution tree representing the service schema to which the instance belongs. After activation, the current node associated with e_j is the root node ε. The instance is ready to execute and responds to the client with the message:

$$e_j\ \ \textbf{started} : \textbf{choose}\ \ a_1||\ldots||a_i||\ldots||a_n$$

The purpose is first to acknowledge the client that the instance is activated and is ready for interaction. Secondly, the client receives the correct service instance identifier, i.e. e_j to interact with. Third, a initial set of choices of possible atomic actions is presented to the client.

Choice: This is the step indicating the interaction between the client and the service instance. To the client the service instance is characterized by its external execution tree in terms of possible actions according to the action alphabet of the service. In each execution step, the service instance offers the client a set of possible actions to execute next:

$$e_j:\ \ \textbf{choose}\ \ a_1||\ldots||a_i||\ldots||a_n$$

where $||$ indicates choice. A client chooses an action and sends the message

$$\textbf{do}\ \ a_i, E_i, e_j$$

indicating that the client solicit the instance e_j of service E_i to perform the action a_i next. After receiving the message, instance e_j executes a_i transparently from the client who is informed only the execution is finish when the instance offers the client to make another choice. If a client of service E_i is another service E which itself acts on behalf of its client, the transparent delegation of client's choice by E to E_i is an advantage because the client does not have to keep track of the execution, E acts as a *server* towards the

client while activating an instance e_j of service E_i and forwards the request to e_j to execute. E becomes a client of e_j and interacts with the *external view* of instance e_j. E decides on which action to execute itself[28] or delegate to another service according to its internal execution tree.

Termination: If the current node on the external execution tree is a final node, i.e. the client is allowed to terminate interaction with the service, the service instance proposes a choice for termination with the action **end** among the evocable actions

$$e_j : \textbf{ choose } end||a_1||\ldots||a_i||\ldots||a_n$$

if the client has reached its goal and decides to terminate, it sends the message:

$$\textbf{do } end, E_i, e_j$$

to deallocate all the resources associated with the instance e_j of service E_i. The service finishes by responding with

$$e_j : \textbf{ ended}$$

to acknowledge termination is successful.
ation with the action **end** among the evocable actions

$$e_j : \textbf{ choose } end||a_1||\ldots||a_i||\ldots||a_n$$

if the client has reached its goal and decides to terminate, it sends the message:

$$\textbf{do } end, E_i, e_j$$

to deallocate all the resources associated with the instance e_j of service E_i. The service finishes by responding with

$$e_j : \textbf{ ended}$$

to acknowledge termination is succesful.

The following examples illustrates scenarios of interactions involving a client, a composite service[29] and component services from a service community C based on the described conceptual interaction protocol. The examples are shown with this setup:

- the client is shown in the left column, it interacts with one instance e of the composite service E;
- the interaction of the composite service with the client and other component services is shown in the middle column;
- in these examples for brevity, there are at most 2 instances of component services simultaneously active whose interactions with the composite service is shown on the right column;
- since execution steps of the different execution trees can be interleaved temporally, it is necessary to number these steps globally to indicate the relative order of execution for the involved services. This is shown on the left side of each column in the examples.

[28] for instance according to the this identifier of its internal execution tree

[29] i.e. one virtual service which exports its behavior as a composite behavior of all component services in the community which it uses to synthesize an overall behavior.

It paves the way to describe service composition and synthesis in the *Roman Model* which is based on the central idea of using a synthesized (virtual) composite service for delegation of actions to available community services and suitably orchestrating the component service instances in the community to execute the delegated actions.

Example 4.3.17. Figure 4.5 [Berardi et al., 2003c] shows an example of interactions between an instance e of service E and its client. E is a (virtual) composite service obtained by executing two service instances, e_1 and e_2 in a non-interleaved way, i.e., either e_1 or e_2 is active no action of other service instances is executed. Full delegation means here E completely delegates the execution of actions to its component services E_1 and E_2 and E itself becomes a *pure orchestrator*. To the component services, the composite service exposes its behavior to them via its external execution tree, therefore its own implementation, i.e., its interaction with the other component services is not divulged to the service community. To the client, composite service instance e forwards the actions offers by e_1 and e_2 transparently without divulging knowledge to the client of which instance of component service it is interacting with. Analogously e forwards the client's request to e_1 and e_2.
A clear interaction between the instances can be studied in steps 1 to 10 where the transparency of interaction is obvious. The client initiates interaction with the composite service E by sending an activation request to E to create an instance e of E. After e has been created it offers the client the actions $end||a_1||a_2$; it is worth to notice that in the initial state (root note ε of the external execution tree of E) the action end is an option because ε is also final, i.e., the client is allowed to terminate without further interaction. The other two offered actions a_1, a_2 are *possible* actions which correspond to the state of the node of the external execution tree of E. E's internal execution tree contains the information about the actions and therefore E 'knows' that service E_1 can execute a_1 from the information of the external execution tree of service E_1, therefore E sends an activation command to E_1 since no instance of E_1 has been created yet. In step 5 instance e_1 of E_1 is created and instance e_1 acknowledges E by offering with its actions for E to choose (notice also in the initial state of E_1, the end action is also possible). Composite service instance e then transparently forwards the client's request for executing action a_1 to instance e_1 in step 6. Notice e transparently substitutes the instance e with e_1 in the client's command indicating the client should and need not know which component service instance actually executes the action. In step 7 e_1 finishes the execution of a_1 and replies to e by offering the actions a_4 or a_5 (according to E_1's external execution tree for allowed actions in that state) for e to choose to execute next. In step 8 e forwards this choice to the client. Since the client only interacts with e, it chooses to execute a_5 and sends doa_5, E, e to e in step 9 who further forward this choice to e_1 by sending doa_5, E_1, e_1 to e_1. In step 10, notice that e 'remembers' the offered actions by e_1 to which it is interacting, therefore it substitutes the instance with e_1 in the command it sends to e_1.
Non-interleaved interaction in this example can be understood as a 'serialization' of interaction between the composite service and the component services where a component service instance must have executed from its activation until its end before another component services can be activated to execute subsequent actions. For instance, in step 12 where component service instance e_1 is active and e must interact with e_1 to termination before interacting with another instance. In the previous step, e_1 has offered the choices either to terminate or execute action a_6 next. Notice that component service instance e_2 is not active in step 12 and e 'knows' from its external execution tree that e_2 can offer the execution of actions $end||a_7$ once e_2 can be created. Therefore e forwards e_1's choices together with the action a_7 (offered by E_2) to the client to choose. In step 13 the client chooses to execute action a_7 of E_2 who is still not activated because e_1 is still active then. Therefore after receiving the client's request in step 14, e must terminate e_1 first; notice the termination of e_1 is possible because it has offered end in step 11 to e. A termination message is sent to e_1 and in step 15 e_1 replies with $ended$ before being evicted. In step 16 e activates an instance of E_2 and forwards the client's request to the component instance e_2 in step 18.

□

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **started:**	5. **started:**
3. **do** a_1, E, e	**choose** $end\|\|a_1\|\|a_2$	**choose** $end\|\|a_1\|\|a_3$
9. **do** a_5, E, e	4. **activate** E_1	7. **choose** $a_4\|\|a_5$
13. **do** a_7, E, e	6. **do** a_1, E_1, e_1	11. **choose** $end\|\|a_6$
21. **do** end, E, e	8. **choose** $a_4\|\|a_5$	15. **ended**
	10. **do** a_5, E_1, e_1	
	12. **choose** $end\|\|a_6\|\|a_7$	
	14. **do** end, E_1, e_1	e-Service E_2: instance e_2
	16. **activate** E_2	17. **started:**
	18. **do** a_7, E_2, e_2	**choose** $end\|\|a_7$
	20. **choose** $end\|\|a_8$	19. **choose** $end\|\|a_8$
	22. **do** end, E_2, e_2	23. **ended**
	24. **ended**	

Figure 4.5: Full delegation with non-interleaved execution: composite service E delegates to at most one active instance e_1 or e_2 of service E_1 or E_2 respectively

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **started:**	7. **started:**
3. **do** a_1, E, e	**choose** $end\|\|a_1\|\|a_2$	**choose** $end\|\|a_1\|\|a_5$
5. **do** a_5, E, e	4. **choose** $a_4\|\|a_5$	9. **choose** $a_6\|\|a_7$
11. **do** a_8, E, e	6. **activate** E_1	17. **choose** $end\|\|a_{12}$
15. **do** a_7, E, e	8. **do** a_5, E_1, e_1	23. **ended**
19. **do** a_{13}, E, e	10. **choose** $a_6\|\|a_7\|\|a_8$	
21. **do** end, E, e	12. **choose** $a_6\|\|a_7\|\|a_{11}$	
	16. **do** a_7, E_1, e_1	
	18. **choose** $a_{12}\|\|a_{13}$	
	20. **choose** $end\|\|a_{12}$	
	22. **do** end, E_1, e_1	
	24. **ended**	

Figure 4.6: Partial delegation with interleaved execution: composite service E executes actions itself or delegates to at most one active instance of a component service

This interaction characterizes services that are *deterministic* because they are *fully controllable*. It means that the execution of a specific operation among those allowed in a certain state will always reach a certain *deterministic* successor state [Berardi et al., 2003a, Berardi et al., 2005c]. By suitably assigning operation execution, it is possible to fully control the transitions of the available services. The *Roman Model* has been extended to handle *non-deterministic* services [Berardi et al., 2005d, Berardi et al., 2006a] which are not fully controllable because the result of interaction with the client sometimes cannot be foreseen due to non-deterministic outcome of execution of component services. Thus the behavior of the available services are only partially controllable. Non-determinism is intrinsic to the nature of real-world services [Berardi et al., 2006b], for instance, the client of such non-deterministic services can invoke them without the ability to control the results they produce.

Example 4.3.18. Figure 4.6 shows an example of interaction with partial delegation and interleaved action execution. Interleaving means that a component service instance does not need to be terminated and partial delegation means that the composite service can choose to either execute a requested action itself or delegate it to a component service instance.

We can notice the effect of partial delegation in step 9 to 12. After an instance e of service E has created component service instance e_1, execution of client's requested actions has been delegated to instance e_1 in the previous steps. In step 9 e_1 has reached a position of its external execution tree where it offers e the actions end, a_1 or a_5 to execute next. Together with its own action a_8 offered by e itself, it offers the client a_8 and the actions of instance e_1 to the client to execute next in step 10. In step 11 the client chooses to execute action a_8 by sending e the command doa_8, E, e. Since a_8 is the executable action of e itself, it does not have to delegate it to the component instance e_1; instead e executes action a_8 immediately and this is an effect of partial delegation by e. It is worth notice that since the client requests to execute a_8 which is handled by e, only e's position in its external execution tree progresses while the current position on the external execution tree of component service instance e_1 does not change. Therefore in the next step, e still offers the same executable actions a_6, a_7 together with its own a_{11} (which reflects the changed current position on e's external execution tree) to the client to choose.
It is also worth mentioning that the current position on the external execution tree of the composite service instance e always changes because beside keeping track of its own offered actions, it must also keep track of those offered by component service instance which in this case is those of e_1. Therefore the step number increments from 12 to step 15 because it must delegate a_7 to e_1 to invoke and keep track of progress in the current position of the execution trees in an interleaved manner.
Another point is the termination issue based on the existence of final node on the external execution tree. Since e is a composite service instance which delegates the execution of actions in an interleaved manner, it can only terminate execution if its own current position of the external execution tree coincides with a final node position of the external execution tree of its component service instance. Put it differently, a position (node) of the external execution tree of e can be final iff the current position is final on all external execution trees of its used component service instances. This characteristics on requirement for termination is obvious in step 18 of this example where e_1 has offered the actions end and a_{12} in the previous step. However since the current position on the external execution tree of e does not coincide with a final node, it cannot offer the client to terminate and therefore it offers only a_{12} (from e_1) together with its possible action a_{13} to the client to choose. We also observe that e has to make e_1 terminate before ending itself in step 24. Another point is the termination issue which is based on the existence of final node on the external execution tree. Since e is a composite service instance which delegates the execution of actions in an interleaved manner, it can only terminate execution if its own current position of the external execution tree coincides with a final node position of the external execution tree of its component service instance. Put it differently, a position (node) of the external execution tree of e can be final iff the current position is final on all external execution trees of its used component service instances. This characteristics on requirement for termination is obvious in step 18 of this example where e_1 has offered the actions end and a_{12} in the previous step. However since the current position on the external execution tree of e does not coincide with a final node, it cannot offer the client to terminate and therefore it offers only a_{12} (from e_1) together with its possible action a_{13} to the client to choose. We also observe that e has to make e_1 terminate before ending itself in step 24.

□

Example 4.3.19. In this example shown in figure 4.7, the interactions involve a composite service instance e that partially delegates actions to a component service instance e_1. In the interaction step 13 to 16 e_1 offers the actions end and a_8 for e while e offers the actions a_8, a_9 or end for the client to choose. The client chooses to execute action a_9. e can offer end because the current position on its external execution tree and that of the component service instance coincides with a final node. In the non-interleaved interaction, e has to first terminate its interaction with e_1 before it can execute its own action a_9. e cannot offer its own action while e_1 is still executing and has to wait for e_1 to offer an end action.

□

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **started:**	7. **started:**
3. **do** a_1, E, e	**choose** $end \| \| a_1 \| \| a_2$	**choose** $end \| \| a_1 \| \| a_5$
5. **do** a_5, E, e	4. **choose** $a_4 \| \| a_5$	9. **choose** $a_6 \| \| a_7$
11. **do** a_7, E, e	6. **activate** E_1	13. **choose** $end \| \| a_8$
15. **do** a_9, E, e	8. **do** a_5, E_1, e_1	17. **ended**
19. **do** end, E, e	10. **choose** $a_6 \| \| a_7$	
	12. **do** a_7, E_1, e_1	
	14. **choose** $end \| \| a_8 \| \| a_9$	
	16. **do** end, E_1, e_1	
	18. **choose** $end \| \| a_{10}$	
	20. **ended**	

Figure 4.7: Partial delegation with non-interleaved execution: composite service E executes actions itself or delegates to at most one active instance of a component service

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **started:**	5. **started:**
3. **do** a_1, E, e	**choose** $end \| \| a_1 \| \| a_2$	**choose** $end \| \| a_1 \| \| a_3$
9. **do** a_1, E, e	4. **activate** E_1	7. **choose** $a_3 \| \| a_4$
15. **do** a_3, E, e	6. **do** a_1, E_1, e_1	17. **choose** $a_5 \| \| end$
19. **do** a_4, E, e	8. **choose** $a_3 \| \| a_4 \| \| a_1$	25. **ended**
23. **do** end, E, e	10. **activate** E_1	
	12. **do** a_1, E_1, e_2	
	14. **choose** $a_3 \| \| a_4$	e-Service E_1: instance e_2
	16. **do** a_3, E_1, e_1	11. **started:**
	18. **choose** $a_5 \| \| a_3 \| \| a_4$	**choose** $end \| \| a_1 \| \| a_3$
	20. **do** a_4, E_1, e_2	13. **choose** $a_3 \| \| a_4$
	22. **choose** $a_5 \| \| end$	21. **choose** end
	24. **do** end, E_1, e_1	27. **ended**
	26. **do** end, E_1, e_2	
	28. **ended**	

Figure 4.8: Partial delegation with interleaved execution: composite service E executes actions itself or delegates to two simultaneously active component service instances

Client C	e-Service E: instance e	e-Service E_1: instance e_1
1. **activate** E	2. **2. started:**	**5. started:**
3. **do** a_1, E, e	**choose** $end \| \| a_1 \| \| a_2$	**choose** $end \| \| a_1 \| \| a_3$
9. **do** a_6, E, e	4. **4. activate** E_1	**7. choose** $a_3 \| \| a_4$
11. **do** a_3, E, e	6. **6. do** a_1, E_1, e_1	**13. choose end**
17. **do** a_1, E, e	8. **8. choose** $a_3 \| \| a_4 \| \| a_6$	**15. ended**
23. **do** a_9, E, e	10. **10. choose** $a_3 \| \| a_4 \| \| a_7$	
25. **do** a_4, E, e	12. **12. do** a_3, E_1, e_1	
29. **do** a_5, E, e	14. **14. do** end, E_1, e_1	e-Service E_1: instance e_2
33. **do** end, E, e	16. **16. choose** $a_1 \| \| a_2 \| \| end$	**19. started:**
	18. **18. activate** E_1	**choose** $end \| \| a_1 \| \| a_3$
	20. **20. do** a_1, E_1, e_2	**21. choose** $a_3 \| \| a_4$
	22. **22. choose** a_9	**27. choose** a_5
	24. **24. choose** $a_3 \| \| a_4 \| \| a_{10}$	**31. choose** $a_6 \| \| end$
	26. **26. do** a_4, E_1, e_2	**35. ended**
	28. **28. choose** a_5	
	30. **30. do** a_5, E_1, e_2	
	32. **32. choose** $a_{11} \| \| end$	
	34. **34. do** end, E_1, e_2	
	36. **36. ended**	

Figure 4.9: Partial delegation with interleaved execution: composite service E delegates to two component service instances with at most one instance being active at a time

Example 4.3.20. Figure 4.8 shows a composite service instance e in interaction with two simultaneously active instances e_1 and e_2 of the same component service E_1. Since both instances has offered to execute action a_3 and both are active in step 13, which component service instance e delegates the execution to will be decided according to the information encoded in e's internal execution tree. When the client chooses to execute the action in step 15, the composite service instance e delegates it to instance e_1 according to the current position on e's internal execution tree in step 16.

□

Example 4.3.21. Figure 4.9 shows a composite service instance e in interaction with two instances e_1 and e_2 of the same component service E_1 that are active one at a time. It is different than the previous example in the sense that over time the external execution trees of both instances e_1 and e_2, despite they belong to the same component service E, will evolve differently in terms of the change in current position on their respective trees. This is due to the fact that the client has made different choices on the actions to execute on them via the solicitation of e. The path from the root to the current position on the external execution trees of the component service instances is different over time. It is worth mentioning that at some point, for instance, in step 21 to 24, e may decide to offer part of or none of the actions of the component services to the client to choose. We observe that e decides to 'mask out' the choices of e_2 of step 21 and offer the client in step 22 only its own action a_9. This is also due to the state transition discussed in the intrinsic characteristics of e's internal execution tree to encode the information of interleaved interaction with the component service instance e_2 which is the single active component service instance momentarily[30].

□

Definition 4.3.22 (Service Orchestrator). Let C be a service community with a set of component services $\{E_1, E_2, \ldots, En\}$, each component service possesses it *action alphabet* Σ_{E_i} and the community C shares

[30] e_1 has been terminated already

a common *community action alphabet* Σ_C with $\Sigma_C = \bigcup_{i=1}^{n} \Sigma_{E_i}$. A *service orchestrator* is a component w.r.t. C which is able to activate, stop and resume each of the available component services using the described *abstract conceptual interaction protocol* and select service instances to perform action execution. It shares the community action alphabet Σ_C and has *full observability* on available service states at runtime by keeping track of the execution states of the component services in C using the information encoded in their *external execution trees*. A *service orchestrator* exposes a desired target behavior to the client and is the target component of service composition. Therefore the *orchestrator* is characterized by:

- its functionality of offering the correct set of actions to the client according to the external schema of the target service;
- delegation of action chosen by the client to the service that offers it;
- termination of execution of service instances belonging to services in the service community.

□

4.3.3 Target Service Specification

The fact that a service community can be coordinated by using an orchestrator component to effectively orchestrate the runtime interaction between a delegating service and a set of delegable services in a community has inspired the notion of using a formal specification to represent a client request w.r.t. the service functionality desired. In case that a client request can be satisfied by a single service which has the functionality enough to cater the client. On the other hand if there is no single service satisfying the requested functionality, then services in the community can be combined and coordinated to satisfy the request. The notion of client specification reflected in the specification of a virtual *target service* to guide functionality synthesis of the available and suitable services. It is thus quintessential to the notion of service composition.

Definition 4.3.23 (Target Service). A *target service* E_{target} is a formal specification of a client request containing the external schema $\mathcal{A}^{E_{ext}}$ of the desired service which the client intends to activate and interaction with.

□

A target service is a *virtual* service which represents an abstract specification of the desired behavior of a service desired by the client. A target service is not executed per se by simply activating an instance of it because there is no *underlying implementation* for it yet. Therefore a target service must be realized by using the available component services in a service community. A *service composer* synthesizes the partial functionality of the component services suitably into a target service schema at design time and uses a *service orchestrator* as a coordination component to steer the component service instances at runtime.

Definition 4.3.24 (Service Composition). Let C be a *well-formed* service community and $\mathcal{A}^{E_{ext}}$ be the external schema of the target service E_{target} expressed in the community alphabet Σ_C of C. A *composition* of E_{target} is an internal schema $\mathcal{A}^{E_{int}}$ of the target service with the following characteristics:

- The internal execution tree $T^{int}(E_{target})$ of the target service *conforms* to its external execution tree $T^{ext}(E_{target})$;
- The internal execution tree $T^{ext}(E_{target})$ fully delegates all actions to the available services in the community C, i.e., the target service employs full delegation and the instance identifier this[31] is not encoded in the target service's internal execution tree $T^{int}(E_{target})$;

[31] it is the identifier to reference the own service (target service in this case)

- $T^{int}(E_{target})$ is *coherent* with C.

□

Corollary 4.3.25 (Composition Existence). *Given a service community C and the external execution tree $T^{int}(E_{target})$ of a target service E_{target} described in definition* (4.3.24)*, composition existence is the problem of checking whether there is a composition of E_{target} w.r.t. C.*

□

Definition 4.3.26 (Composition Synthesis). Given a service community C and the external execution tree $T^{int}(E_{target})$ of a target service E_{target}, a *composition synthesis* is the process of synthesizing an internal schema $\mathcal{A}^{E_{int}}$ for E_{target} which realizes the composition of E_{target} w.r.t. C. It is characterized by dualistic manifestation of:

- the external schema $\mathcal{A}^{E_{ext}}$ expressed in a suitable formalism[32] and represented by $T^{ext}(E_{target})$ as the exported behavior of the target service;
- the internal schema $\mathcal{A}^{E_{int}}$ expressed in a suitable formalism and represented by $T^{int}(E_{target})$ realizing and implementing the composition w.r.t. C using the concept of a *service orchestrator* defined in definition (4.3.22).

□

4.3.4 Automatic Composition Synthesis

Finite state automata have been adopted in academia [Bultan et al., 2003, Mecella et al., 2004] as a means to model the exported conversational behavior of services since their external and internal schema can be expressed with finite number of states using deterministic finite state machines (FSM).

Behavior Description with Finite State Machine

As a guiding example for the following description of service composition, we introduce a target service E_0 as described in definition (4.3.23) as a client specification of the desired functionality. Here the target is represented as a deterministic FSM which is shown is figure 4.10. This example target service is based on a simplified view of an stock investment and quoting service. Though many services on the internet support only one operation per service, we call these services monolithic. There are nevertheless many multi-operational services which support more than one (often of a process oriented nature) operation in each service. We can also model the former monolithic type of services using the finite state machine approach easily since the prerequisite of number of states and state transitions as well as number of atomic operation are very limited (usually one action and two states per service). Therefore it is our modeling of the latter type of services using deterministic finite state machine that can unleash the full potential of the FSM modeling of these services because they can underly complex processes which support many operations and therefore require considering more atomic actions and larger number of states transitions in the model. In this example the service operations (modeled as labeled and abbreviated atomic actions) of the target service are:

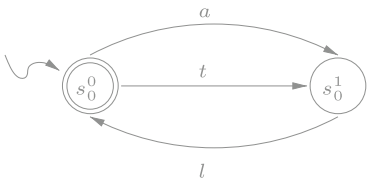

Figure 4.10: FSM external schema $\mathcal{A}^{ext}(E_0)$ of the target service E_0

[32] i.e. a formal language

1. label "a" is a shorthand for the atomic action request_up_to_date_stock_quote in which a client can ask for current stock quote data regarding a specific stock;
2. label "t" is a shorthand for the atomic action request_stock_quote_history in which a client can request past stock quote data regarding a certain stock;
3. label "l" is a shorthand for the atomic action list_and_chart_data which displays the requested data in charts on the broker's terminal screen.

The action alphabet of the target service E_0 contains these atomic actions $\Sigma = \{a, t, l\}$. Beside the atomic actions, we notice that the states of the FSM are represented as nodes with subscript indicating affiliation to a specific service. Conventionally the subscript "0" for instance in E_0 is used to indicate the target service while the subscripts 1 to n are used to indicate services in a community of size n. Indices of the state notation is used to enumerate the different states of a finite state machine starting from 0. Every real world service is and can be modeled with a finite set of states since there exists no service consisting of infinite number of states. Double circular node denotes final states meaning that interaction with the service can be terminated in that state, for instance, the FSM of E_0 contains the state s_0^0 which is incidentally the start state as well as a final state in figure 4.10. Notice that a start state does not need to be a final state and a final state does not always coincide with a start state. There are two component services E_1 and E_2 in the community that are available and contains relevant operations which can satisfy our purpose.

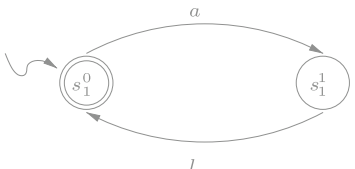

Figure 4.11: FSM external schema $\mathcal{A}^{ext}(E_1)$ of component service E_1

The external schemas of service E_1 and E_2 are shown in figure 4.11 and 4.12 respectively. Service E_1 offers the operations request_up_to_date_stock_quote as its directed edge indicates. From state s_1^1 the outbound edge of E_1 labeled with l offers the operation list_and_chart_data to chart the requested data. This sequence of operations of E_1 is reasonable and represents a service semantics that E_1 is intended for listing and charting current stock quotes. Service E_2 offers the operations to execute request_stock_quote_history first to request a history of past performance of a certain stock and then to chart the history data, supporting the semantics of a history-based view of the performance of a company's stocks. Since the target service specifies a request for the functionality of both obtaining the current data as well as a historic performances of a specific company's stock, none of these community services can satisfy our request alone. However they constitute the necessary functionality which can be synthesized and coordinated so that the request can be satisfied.

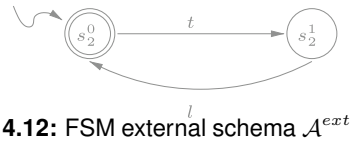

Figure 4.12: FSM external schema $\mathcal{A}^{ext}(E_2)$ of component service E_2

Definition 4.3.27 (FSM External Schema). Let C be a service community with the *community action alphabet* Σ_C and E be a service in C. The external schema of E is a *finite state machine* FSM $\mathcal{A}^{E_{ext}} = (\Sigma_C, S_E, s_E^0, \delta_E, F_E)$ with:

- Σ_C is the alphabet of the FSM; since E is in C its FSM has the common alphabet as the community action alphabet;
- S_E is the finite set of states of the FSM, representing the finite set of states of E;
- s_E^0 is the initial state of the FSM, representing the initial state of E;
- $\delta_E : S_E \times \Sigma_C \rightarrow S_E$ is the partial transition function of the FSM, that when given a state s with $s \in S_E$ and an action a with $a \in \Sigma_C$ returns the successor state resulting from execution of a in the state s;

- $F_E \subseteq S_E$ is the set of final states of the FSM, i.e. the set of legal states where the interaction with E can be terminated.

For brevity the FSM external schema $\mathcal{A}^{E_{ext}}$ can be abbreviated and written as $\mathcal{A}^{ext}(E)$ where $E \in C$.

□

Corollary 4.3.28 (FSM External Execution Tree). *Let $\mathcal{A}^{ext}(E)$ be the FSM external schema of a service E in community C as described in definition* (4.3.27) *and $\theta(\cdot)$ be an auxiliary state mapping function. An FSM external schema $\mathcal{A}^{ext}(E)$ can be unfolded into an FSM external execution tree $T(\mathcal{A}^{ext}(E))$ by assigning each node of the tree a state of the FSM using the auxiliary state mapping function $\theta(\cdot)$. We contruct $T(\mathcal{A}^{ext}(E))$ by:*

- *assigning the root node ε of $T(\mathcal{A}^{ext}(E))$ the initial state s_E^0 using $\theta(\varepsilon)$;*
- *given a node x in $T(\mathcal{A}^{ext}(E))$ that is assigned the state s with $s \in S_E$ using $\theta(x)$, then for the action a that is possible and executable from that state s, we define the successor state s' of executing a using the transition function $s' = \delta_E(s, a)$ and assign s' to the next node on the trajectory by $\theta(x \cdot a) = s'$;*
- *assigning a node x the final state iff $\theta(x) \in F_E$.*

□

The FSM external execution tree $T(\mathcal{A}^{ext}(E_0))$ or abbreviated $T(\mathcal{A}_0)$ of our taget service E_0 is obtained by unfolding its FSM external schema shown in figure 4.10 into an external execution tree as described in definition (4.3.8) using the auxiliary mapping function $\theta(\cdot)$ to assign each node a state along every execution trajectory:
$\theta(\varepsilon) = s_0^0$;
$\theta(a) = \theta(t) = s_0^1$;
$\theta(a \cdot l) = \theta(t \cdot l) = s_0^0$;
$\theta(a \cdot l \cdot a) = \theta(a \cdot l \cdot t) = \theta(t \cdot l \cdot a) = \theta(t \cdot l \cdot t) = s_0^1$;
...
Figure 4.13 shows the external execution tree of the target service E_0.

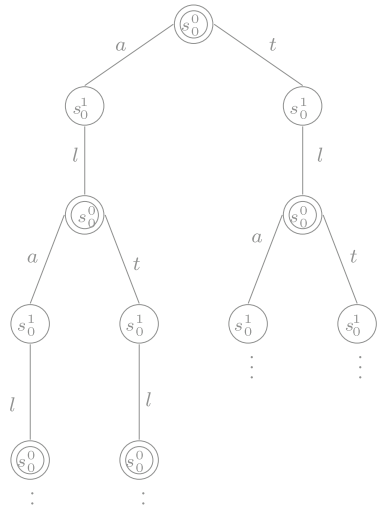

Figure 4.13: Target service external execution tree $T^{ext}(\mathcal{A}_0)$

Definition 4.3.29 (Mealy FSM Internal Schema). Let C be a service community whose community action alphabet is Σ_C and E be a service in C. The *Mealy internal schema* of E is a *Mealy finite state machine*[33] (MFSM) $\mathcal{A}^{E_{int}} = (\Sigma_C, \sigma_C, S_E^{int}, s_E^{0\ int}, \delta_E^{int}, \omega_E^{int}, F_E^{int})$ with:

- $\Sigma_C, S_E^{int}, s_E^{0\ int}, \delta_E^{int}, F_E^{int}$ is defined analogously with the meaning of definition (4.3.27);
- σ_C is the *output alphabet* of the Mealy finite state machine MFSM that is used to denote which services execute each action;
- $\omega_E^{int}(s, a) : S_E^{int} \times \Sigma_C \rightarrow \sigma_C$ is the *output function* of the MFSM; given a state s with $s \in S_E$ and an action a with $a \in \Sigma_C$ the output function returns a subset of services in C in terms of *output alphabet* σ_C that can execute action a when the service E is in state s. If the returned subset is empty, then this is returned implying that the service E executes a itself. It is assumed that ω_E^{int} is defined if δ_E^{int} is defined.

[33] A Mealy finite state machine is defined as a finite state machine with output.

For brevity the MFSM internal schema can be abbreviated to $\mathcal{A}^{int}(E)$ where $E \in C$.

□

Figure 4.14 shows an MFSM internal schema for the example target service E_0 as depicted in figure 4.10. With the community action alphabet Σ_C consisting of the union of executable atomic actions a, t, l, its output function $\omega_E^{int}(s, a)$ returns the delegable service as follows:
$\omega_E^{int}(s_0^0, a) = \{1\}$ means action "a" is delegable to E_1 in state s_0^0;
$\omega_E^{int}(s_0^1, l) = \{1\}$ means action "l" is delegable to E_1 in state s_0^1;
$\omega_E^{int}(s_0^0, t) = \{2\}$ means action "t" is delegable to E_2 in state s_0^0;
$\omega_E^{int}(s_0^2, l) = \{2\}$ means action "l" is delegable to E_2 in state s_0^2.
In fact the output function $\omega_E^{int}(s, a)$ is defined if the corresponding transitions w.r.t. the transition function δ_E^{int} is defined. Here the MFSM can be viewed to have two main delegable trajectories: one which is realized by the executable actions of the component service E_1 and the other by E_2 which together synthesize the functionality of the target service.

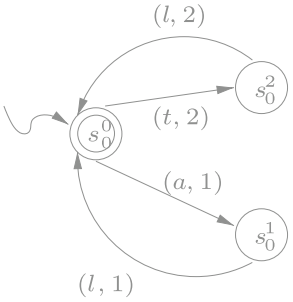

Figure 4.14: MFSM internal schema $\mathcal{A}^{int}(E_0)$ of target service E_0

Corollary 4.3.30 (MFSM Internal Execution Tree). *Let $\mathcal{A}^{int}(E)$ be the Mealy FSM internal schema of a service E in community C as described in definition* (4.3.29) *and $\theta^{int}(\cdot)$ be an auxiliary state mapping function. The Mealy FSM internal schema $\mathcal{A}^{int}(E)$ can be unfolded into an Mealy FSM internal execution tree $T(\mathcal{A}^{int}(E))$ by assigning each node of the tree a state of the Mealy FSM using the auxiliary state mapping function $\theta^{int}(\cdot)$. We contruct $T(\mathcal{A}^{int}(E))$ by:*

- *assigning the root node ε of $T(\mathcal{A}^{int}(E))$ the initial state $\theta^{int}(\varepsilon) = s_E^{0\,int}$;*
- *given a node x in $T(\mathcal{A}^{int}(E))$ that is assigned the state s with $s \in S_E^{int}$ using $\theta^{int}(x)$, then for the action a that is possible and executable from that state s, we define the successor state s' of executing a using the transition function $s' = \delta_E^{int}(s, a)$ and assign s' to the next node on the trajectory by $\theta^{int}(x \cdot a) = s'$;*
- *labeling each edge in $T(\mathcal{A}^{int}(E))$ with a pair (a, E_i) by using the output function $\omega_E^{int}(s, a) = \{E_i\}$ to obtain the subset E_i with $E_i \in \sigma_C$ to indicate which subset of services to delegate the action a to when the Mealy FSM is in state s;*
- *assigning a node x the final state iff $\theta^{int}(x) \in F_E^{int}$.*

□

Figure 4.15 shows the internal execution tree of our target service illustrated in figure 4.10 by induction on the node level. The edges are labeled with tuples (a, n) where "a" is an atomic action and "n" with $n \in \mathbb{N}$ the identifier of the delegable service. This labeling scheme resembles that of an abstract internal execution tree as described in definition (4.3.9). In the abstract labeling schema the emphasis is on the instance level delegation where the edges are labeled with pairs of (a, I), i.e. an action "a" is delegable to a tuple I representing an instance that belongs to a service. Here in the MFSM internal execution tree, it is only necessary to represent delegable service with its identifier which is denoted by "n". We observe that an FSM external schema and its corresponding Mealy FSM internal schema sometimes may have different topological structures and therefore the Mealy FSM

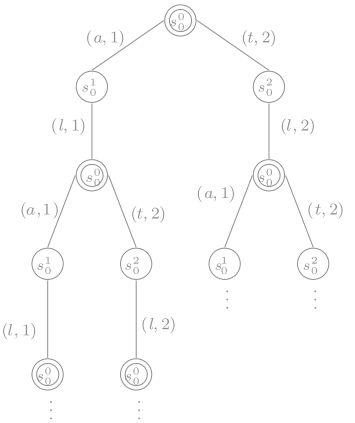

Figure 4.15: Target service internal execution tree $T^{int}(\mathcal{A}_0)$

internal schema cannot be obtained by simply labeling the FSM external schema with services in a service community.

Deterministic Propositional Dynamic Logic

Based on the previous description of service behavior of a composite service using finite state machine FSM and Mealy finite state machine MFSM, it can be shown that a composition exists for the composite service if there is a Mealy internal schema that is constituted by a MFSM for the composite service. The basic idea is to use a *modal logic* formalism[34] called *deterministic propositional dynamic logic* (DPDL) [Perrin et al., 1990, Fischer & Ladner, 1979, van Emde Boas et al., 1990] to *reduce* the problem of finding a composition to the process of formulation of a suitable DPDL logic formula and checking *satisfiability* of it, in order to prove existence of a composition and synthesize this composition for the composite service.

Deterministic propositional dynamic logic DPDL formulas are built from a set of *atomic propositions* denoted with $\mathcal{P}$ and a set of *deterministic atomic actions* denoted with $\mathcal{A}$. The syntax of the logic is as follows:

$$\begin{array}{lcl} \phi & \longrightarrow & \mathsf{true} \mid \mathsf{false} \mid P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle r \rangle \phi \mid [r]\phi \\ r & \longrightarrow & a \mid r_1 \cup r_2 \mid r_1; r_2 \mid r^* \mid \phi? \end{array}$$

where ϕ is a DPDL atomic logical formula, P is an atomic proposition in $\mathcal{P}$ and a is an atomic action in $\mathcal{A}$ with r as a *regular expression* over the set of actions in $\mathcal{A}$. The logical operators have their conventional meaning, for instance, "$\neg$" is used for negation of a formula, binary operator "$\wedge$" is a logical connective used for logical-AND conjunction of two DPDL formulas and "$\vee$" is used for logical-OR conjunction of two formulas. For logical implication we can rewrite the logical expression from $\phi_1 \rightarrow \phi_2$ to $\neg\phi_1 \vee \phi_2$ as a shorthand. The two special propositional atomic terms `true` and `false` are used to represent the constant truth and falsehood of a proposition respectively.
The modal logic operator $\langle r \rangle \phi$ means that there exists an execution of r reaching a state where the DPDL formula ϕ holds[35]. The other modal logic operator $[r]\phi$ universally quantifies over r when evaluating ϕ stating that all terminating executions of r reach a state where ϕ holds. The binary connective "$\cup$" in $r_1 \cup r_2$ means choosing non deterministically between r_1 or r_2, while ";" has a meaning of sequential execution in $r_1; r_2$ stating that the execution of r_1 must precede r_2. The expression r^* means that r is executed a non deterministically chosen number of times (zero or more) and $\phi?$ means that the condition expressed in the DPDL formula ϕ must be tested and if the test succeeds, execution can proceed or otherwise not.
The semantics of a DPDL formula is based on the foundation of *deterministic Kripke structure*. A deterministic Kripke structure denoted with $\mathcal{I}$ is a tuple of the form $\mathcal{I} = (\triangle^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \mathcal{A}}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$, where $\triangle^{\mathcal{I}}$ denotes a non-empty set of states called *worlds*; $\{a^{\mathcal{I}}\}_{a \in \mathcal{A}}$ is a family of partial functions $a^{\mathcal{I}} : \triangle^{\mathcal{I}} \rightarrow \triangle^{\mathcal{I}}$ mapping from a definition domain of $\triangle^{\mathcal{I}}$ to a value domain of $\triangle^{\mathcal{I}}$ which specifies the state transition caused by the atomic action a; the third component of the Kripke structure $\{P^{\mathcal{I}}\}_{P \in \mathcal{P}} \subseteq \triangle^{\mathcal{I}}$ denotes all the propositional elements $P^{\mathcal{I}}$ of $\triangle^{\mathcal{I}}$ where P is true.
Logical entailment defines the semantics of the Kripke structure which states a formula ϕ holds (true) at a state s within a Kripke structure $\mathcal{I}$. By logical induction on ϕ, entailment $\mathcal{I}, s \models \phi$ is based on *interpretation* of ϕ using Kripke structure $\mathcal{I}$. The interpretation semantics of DPDL propositions is shown as follows:

[34] In *modal logic* a *modality* is a connective which takes a formula or a set of formulas and produces a new formula with a new meaning. The semantics of DPDL formulas must be defined over a Kripke structure because of modality.
[35] The modal logic operator $\langle r \rangle \phi$ existentially quantifies over r when evaluating ϕ.

$$
\begin{array}{ll}
\mathcal{I}, s \models \texttt{true} & \top \\
\mathcal{I}, s \models \texttt{false} & \bot \\
\mathcal{I}, s \models P & \text{iff } s \in P^{\mathcal{I}} \\
\mathcal{I}, s \models \neg\phi & \text{iff } \mathcal{I}, s \not\models \phi \\
\mathcal{I}, s \models \phi_1 \wedge \phi_2 & \text{iff } \mathcal{I}, s \models \phi_1 \text{ and } \mathcal{I}, s \models \phi_2 \\
\mathcal{I}, s \models \phi_1 \vee \phi_2 & \text{iff } \mathcal{I}, s \models \phi_1 \text{ or } \mathcal{I}, s \models \phi_2 \\
\mathcal{I}, s \models \langle r \rangle \phi & \text{iff } \exists s' \text{ such that } (s, s') \in r^{\mathcal{I}} \text{ and } \mathcal{I}, s' \models \phi \\
\mathcal{I}, s \models [r] \phi & \text{iff } \forall s', (s, s') \in r^{\mathcal{I}} \rightarrow \mathcal{I}, s' \models \phi
\end{array}
$$

where for the executable regular expressions r constituted by a, the second component of the Kripke structure $\{a^{\mathcal{I}}\}_{a \in \mathcal{A}} : \triangle^{\mathcal{I}} \rightarrow \triangle^{\mathcal{I}}$ is extended to include $r^{\mathcal{I}}$ defined by induction on the form of r:

$$
\begin{array}{lcl}
(r_1 \cup r_2)^{\mathcal{I}} & = & r_1^{\mathcal{I}} \cup r_2^{\mathcal{I}} \\
(r_1; r_2)^{\mathcal{I}} & = & r_1^{\mathcal{I}} \circ r_2^{\mathcal{I}} \\
(r^*)^{\mathcal{I}} & = & (r^{\mathcal{I}})^* \\
(\phi?)^{\mathcal{I}} & = & \{(s, s') \in \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}} \mid \mathcal{I}, s \models \phi\}
\end{array}
$$

where an *interpretation* of the Kripke structure over a non-deterministic choice of execution of r_1 and r_2 in $(r_1 \cup r_2)^{\mathcal{I}}$ can be view as the non-deterministic choice of the interpretation of the respective individual execution as expressed in $r_1^{\mathcal{I}} \cup r_2^{\mathcal{I}}$; analogously the sequential execution has a corresponding interpretation of the individual sequential execution threads r_1 and r_2 shown with the connective "$\circ$" in the above expression. The non-deterministically chosen number of iterative execution of r is the interpretation of the execution of each iteration cycle.
Given a DPDL formula ϕ, the Kripke structure $\mathcal{I} = (\triangle^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \mathcal{A}}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$ is a *model* of ϕ if there exists a state $s \in \triangle^{\mathcal{I}}$ such that $\mathcal{I}, s \models \phi$.
The logical semantics of inference is described as follows:

- a formula ϕ is *satisfiable* if there is a *model* of phi and vice versa if there is no *model*, the formula ϕ is *unsatisfiable*;
- a formula ϕ is *valid* in its defining Kripke structure $\mathcal{I}$ if $\forall s \mid s \in \triangle^{\mathcal{I}} \mid \mathcal{I}, s \models \phi$, i.e., all the *models* of $\mathcal{I}$ is also *model* of ϕ;
- *axioms* are formulas which are used to select the subset of formulas of interpretational interest. A Kripke structure $\mathcal{I}$ is a *model* of an *axiom* ϕ' if ϕ' is *valid* in $\mathcal{I}$;
- a Kripke structure $\mathcal{I}$ is a *model* of a finite set of axioms Γ with $\phi' \in \Gamma$ if $\mathcal{I}$ is a *model* of all axioms in Γ, therefore an axiom is *satisfiable* if it has a *model*;
- we observe that a finite set of axioms Γ *entails* or *logically implies* a formula ϕ, written as $\Gamma \models \phi$ if ϕ is *valid* in every model of Γ.

Definition 4.3.31 (DPDL Tree Model Property). Let ϕ be a deterministic propositional dynamic logic (DPDL) formula, Γ be a set of relevant DPDL domain axioms and let there exists a *model* of ϕ such that $\Gamma \models \phi$. The *tree model property* of a DPDL formula states that every *model* of phi can be unwound (unfolded) to a tree-shaped model consisting of domain elements as nodes and (partial) transition functions interpreting actions as edges.

□

Definition 4.3.32 (DPDL Small Model Property). Let ϕ be a deterministic propositional dynamic logic (DPDL) formula, Γ be a set of relevant DPDL domain axioms and let there exists a *model* of ϕ such that $\Gamma \models \phi$. The *small model property* of a DPDL formula states that every *satisfiable* formula admits a finite *model* which size such as the number of domain elements is at most exponential to the size of the formula itself.

□

Corollary 4.3.33 (DPDL Satisfiability Checking Complexity). *From the small model property described in definition* (4.3.32)*, we derive the complexity of checking satisfiability of a deterministic propositional dynamic logic (DPDL) formula is EXPTIME-complete [Ben-Ari et al., 1982].*

□

Existence of Composition

In order to derive a composition if one exists, we proceed by formulating the problem of composition existence to *reduction* into a suitable deterministic propositional dynamic logical (DPDL) formula Φ which is built by conjunction of a set of atomic propositions $\mathcal{P}$ in DPDL and subsequently checking *satisfiability* of Φ w.r.t. a target service E_{target} and a service community $C = \{E_1, \ldots, E_n\}$ consisting of n component services.

Lemma 4.3.34 (DPDL Atomic Propositions Encoding). *Let E_{target} be a target service in the sense described in definition* (4.3.23) *with the FSM external schema $\mathcal{A}^{ext}(E_{target})$ which is abbreviated to A_0 for brevity and C be a service community consisting of n component services whose FSM external schemas are $A_1, \ldots, A_n$ respectively. For describing the states and state transitions of an FSM, a set of atomic DPDL propositions $\mathcal{P}$ consisting of individual atomic proposition Ψ is built which contains:*

- *one state proposition s_j for each state s_j in the FSM service schemas $\mathcal{A}_j$ with $j = 0, \ldots, n$ of each service (target service and the component services), denoting whether the respective service is in state s_j;*
- *final propositions F_j with $j = 0, \ldots, n$ for all FSM service schemas, denoting whether $\mathcal{A}_j$ is in a final state;*
- *transitional propositions moved_j with $j = 0, \ldots, n$ for all FSM service schemas, denoting whether a $\mathcal{A}_j$ has performed a transition within the FSM.*
- *a set of possible atomic actions $\mathbb{A}$ for all FSM service schemas where $\mathbb{A}$ coincide with the community action alphabet Σ_C and the action alphabet Σ_{E_0} of the target service, i.e., $\mathbb{A} = \Sigma = \Sigma_C \cup \Sigma_{E_0}$.*

□

Example 4.3.35. This example illustrates the encoding of atomic propositions with the guiding examples shown in the figures 4.10, 4.11 and 4.12 of section 4.3.4 which introduce the encoding of the set of propositions $\mathcal{P}$. With regard to lemma (4.3.34), $\mathcal{P}$ is the set consisting of:

$$\mathcal{P} = \left\{ s_0^0, s_0^1, s_1^0, s_1^1, s_2^0, s_2^1, F_0, F_1, F_2, \mathsf{moved}_1, \mathsf{moved}_2 \right\}$$

The states s_x^y with $x = 0, 1, 2$ and $y = 0, 1$ directly correspond to the states of the FSM external schemas of the target service and component services. The set of all final states F_0, F_1 and F_2 for the mentioned services which correspond to the double circular nodes of the FSM of the services. The predicate moved_i with $i = 1, 2$ specifies the transitions of the component services stating that only either one service performs a transition in a single step. Finally the set of atomic actions belongs to the action alphabet $\Sigma = \{a, t, l\}$ with "a" being a shorthand for request_up_to_date_stock_quote, "t" for request_stock_quote_history and "l" for list_and_chart_data.

Definition 4.3.36 (DPDL Formula Encoding for Composition). Let E_{target} be a *target service* with the *FSM external schema* $\mathcal{A}^{ext}(E_{target})$ which is abbreviated to A_0 for brevity; let C be a service community consisting of n component services whose FSM external schemas are $A_1, \ldots, A_n$ respectively and Σ be the *community action alphabet* which is also shared by the target service; let $\mathcal{P}$ be a set of atomic DPDL propositions. For formal encoding of the service composition problem, a DPDL formula Φ consisting of a finite number of $\mathcal{P}$ with *master modality*[36] is obtained by *conjunction* of the following DPDL formulas:

- a set of formulas denoted by Φ_{A_0} which encode the target service external schema $A_0 = (\Sigma, S_0, s_0^0, \delta_0, F_0)$;
- sets of formulas denoted by Φ_{A_i} each set for a component service, encoding each component FSM schema $A_i = (\Sigma, S_i, s_i^0, \delta_i, F_i)$;
- formulas denoted by $\Phi_{independent}$ which encode domain independent conditions which specify general universal assertions,

so that $\Phi = \Phi_{A_0} \wedge (\bigwedge_{i=1}^{n} \Phi_{A_i}) \wedge \Phi_{independent}$.

□

The individual component DPDL formulas constituting the main formula Φ will be described in the following. Each Φ consists of a set of DPDL atomic proposition Ψ. We use the master modality "u" in Ψ to encode the *states* and *state transitions* of the FSM which underlies each FSM external schema. Each formula Ψ has the general form $[a]\,\Psi$ and $\langle a \rangle\,\Psi$[37] where a is an atomic action with $a \in \Sigma$ and *master modality* $u = (a_1 \cup \ldots \cup a_m)^*$.

Definition 4.3.37 (DPDL Formula for Target Service). Let Φ_{A_0} be the set of DPDL formulas encoding the *target service* which contains individual atomic propositions Ψ w.r.t. the FSM external schema A_0 of the target service. Φ_{A_0} contains the following Ψ propositions:

- $[u]\,(s \rightarrow \neg s')$ for all *pairs of states* of the FSM: $s \in S_0$ and $s' \in S_0$ with $s \neq s'$; this is important for encoding the *disjointness of states* of the FSM stating that no pair of states can be true simultaneously in the same FSM;
- $[u]\,(s \rightarrow \langle a \rangle\,\mathsf{true} \wedge [a]\,s')$ is the modal transition proposition defined over the transition function $s' = \delta_0(s, a)$ of the FSM specifying the transition of executing a and reaching successor state s';
- $[u]\,(s \rightarrow [a]\,\mathsf{false})$ is the prohibition of transition definition which specifies given a state s and action a such that $\delta_0(s, a)$ is not defined;
- $[u]\,(F_0 \leftrightarrow \bigvee_{s \in F_0} s)$ specifying the final states of FSM, i.e., any legal state s that belongs to F_0.

□

Example 4.3.38. We illustrate definition (4.3.37) to encode our target service as shown in figure 4.10 by using the *master modality* $u = (a \cup t \cup l)^*$ to unify the atomic actions. Recalling that the master modality is used in combination of a DPDL proposition ϕ such that $[u]\,\phi$ whether the proposition holds when an action is executed (among those defined in the master modality). First we must define the disjointness of states:

$$[u]\,(s_0^0 \rightarrow \neg s_0^1)$$

[36] The *master modality* "$[u]$" represents a reflexive transitive closure of the union of all atomic actions belonging to the action alphabet Σ of all services, i.e., $\forall a_i \in \Sigma; u = (\bigcup_{i=1}^{m} a_i)^*$ or $u = (a_1 \cup \ldots \cup a_m)^*$ with the Kleene star "$*$" denoting the master modality to encode universal assertions in the domain of interest.

[37] The $\langle a \rangle$ modality is used only in atomic proposition true.

We then encode possible transitions for the target service:

$$[u]\,(s_0^0 \rightarrow \langle a\rangle\,\mathsf{true} \wedge [a]\,s_0^1)$$

$$[u]\,(s_0^0 \rightarrow \langle t\rangle\,\mathsf{true} \wedge [t]\,s_0^1)$$

$$[u]\,(s_0^1 \rightarrow \langle l\rangle\,\mathsf{true} \wedge [l]\,s_0^0)$$

Subsequently the impossible transitions, i.e., those that are undefined must be encoded:

$$[u]\,(s_0^0 \rightarrow [l]\,\mathsf{false})$$

$$[u]\,(s_0^1 \rightarrow [a]\,\mathsf{false})$$

$$[u]\,(s_0^1 \rightarrow [t]\,\mathsf{false})$$

where the total number of propositions for defined and undefined state transitions is usually equal. Final state for the target service is encoded by:

$$[u]\,(F_0 \leftrightarrow s_0^0)$$

Definition 4.3.39 (DPDL Formula for Component Services). Let Φ_{A_i} be the set of DPDL formulas consisting of atomic propositions Ψ which encodes a *component service* in the community C with size n, i.e. $i = 1, \ldots, n$ w.r.t. the FSM external schema A_i. The set Φ_{A_i} consists of the following Ψ propositions:

- $[u]\,(s \rightarrow \neg s')$ for all *pairs of states* of the FSM: $s \in S_0$ and $s' \in S_0$ with $s \neq s'$, this formula specifies analogously the disjointness of states of an FSM;
- $[u]\,(s \rightarrow [a]\,(\mathsf{moved}_i \wedge s' \vee \neg\mathsf{moved}_i \wedge s))$ encodes transition proposition semantics of the FSM with $s' = \delta_i(s, a)$; it specifies that A_i is required either to make a transition of state if a is executed or the original state s must hold and no transition takes place;
- $[u]\,(s \rightarrow [a]\,(\neg\mathsf{moved}_i \wedge s))$ specifies the state when transition is not defined and therefore a in this state is not executed;
- $[u]\,(F_i \leftrightarrow \bigvee_{s \in F_i} s)$ specifies the final states of FSM external schema A_i.

□

Example 4.3.40. We illustrate with this example the definition (4.3.39) encoding the external schema of our example component services shown in the figures 4.11 and 4.12. For component service E_1 analogous to the previous example we first encode the state disjointness propositions:

$$[u]\,(s_1^0 \rightarrow \neg s_1^1)$$

With account on state transition using the predicate moved_i where $i = 1, 2$ to encode the required state transitions with the semantic that only one component service performs a transition in a single step and that in each step at least one step must be proceeded. For E_1:

$$[u]\,(s_1^0 \rightarrow [a]\,(\mathsf{moved}_1 \wedge s_1^1 \vee \neg\mathsf{moved}_1 \wedge s_1^0))$$

$$[u]\,(s_1^1 \rightarrow [l]\,(\mathsf{moved}_1 \wedge s_1^0 \vee \neg\mathsf{moved}_1 \wedge s_1^1))$$

Similar to the previous example we encode propositions specifying the undefined transitions for E_1:

$$[u]\,(s_1^0 \rightarrow [l]\,(\neg\mathsf{moved}_1 \wedge s_1^0))$$

$$[u]\,(s_1^0 \rightarrow [t]\,(\neg\mathsf{moved}_1 \wedge s_1^0))$$

$$[u]\,(s_1^1 \rightarrow [a]\,(\neg \mathsf{moved}_1 \wedge s_1^1))$$

$$[u]\,(s_1^1 \rightarrow [t]\,(\neg \mathsf{moved}_1 \wedge s_1^1))$$

Finally for E_1 final state is asserted in state s_1^0 and encoded in the proposition:

$$[u]\,(F_1 \leftrightarrow s_1^0)$$

For parsimonious reason, we give the propositions briefly for component service E_2 here in a group:

$$\begin{array}{lcl}
[u]\,(s_2^0 & \rightarrow & \neg s_2^1) \\
[u]\,(s_2^0 & \rightarrow & [t]\,(\mathsf{moved}_2 \wedge s_2^1 \vee \neg \mathsf{moved}_2 \wedge s_2^0)) \\
[u]\,(s_2^1 & \rightarrow & [l]\,(\mathsf{moved}_2 \wedge s_2^0 \vee \neg \mathsf{moved}_2 \wedge s_2^1)) \\
[u]\,(s_2^0 & \rightarrow & [l]\,(\neg \mathsf{moved}_2 \wedge s_2^0)) \\
[u]\,(s_2^0 & \rightarrow & [a]\,(\neg \mathsf{moved}_2 \wedge s_2^0)) \\
[u]\,(s_2^1 & \rightarrow & [t]\,(\neg \mathsf{moved}_2 \wedge s_2^1)) \\
[u]\,(s_2^1 & \rightarrow & [a]\,(\neg \mathsf{moved}_2 \wedge s_2^1)) \\
[u]\,(F_2 & \leftrightarrow & s_2^0)
\end{array}$$

Definition 4.3.41 (DPDL Formula for Domain Independent Conditions). Let $\Phi_{A_{independent}}$ be the set of DPDL formulas consisting of atomic propositions Ψ which encodes domain independent conditions and general universal assertions about the domain. It consists of the following Ψ propositions:

- $s_0^0 \wedge (\bigwedge_{i=1}^{n} s_i^0)$ specifies that all services are in their initial state before any interaction begins;
- $\forall a \in \Sigma$ at least one transition step has proceeded (has moved) in every execution of an action a in the FSM; this condition is encoded with $[u]\,(\langle a \rangle \,\mathsf{true} \rightarrow [a] \bigvee_{i=1}^{n} \mathsf{moved}_i)$;
- $[u]\,(F_0 \rightarrow \bigwedge_{i=1}^{n} F_i)$ specifies the target service is in a final state if all component services are in a final state.

□

Example 4.3.42. Domain independent conditions are the constraints which must hold not only for single services but for the overall composition. These constraints for our composition problem regarding the example services shown in the figures 4.10, 4.11 and 4.12 are encoded as DPDL propositions:

$$s_0^0 \wedge s_1^0 \wedge s_2^0$$

This proposition specifies that all services start in the initial state. On the other hand final state of the target service implies that the component services must be in final state also:

$$[u]\,(F_0 \rightarrow F_1 \wedge F_2)$$

These last propositions encode the fact that in each single processing step at least one service performs a transition:

$$\begin{array}{lcl}
[u]\,(\langle a \rangle \,\mathsf{true} & \rightarrow & [a]\,(\mathsf{moved}_1 \vee \mathsf{moved}_2)) \\
[u]\,(\langle t \rangle \,\mathsf{true} & \rightarrow & [t]\,(\mathsf{moved}_1 \vee \mathsf{moved}_2)) \\
[u]\,(\langle l \rangle \,\mathsf{true} & \rightarrow & [l]\,(\mathsf{moved}_1 \vee \mathsf{moved}_2))
\end{array}$$

Reduction of the service composition problem to a DPDL formula Φ makes it possible to *check satisfiability* of this formula which contains propositions about the dynamic aspects of the finite state transitions systems encoded with their actions, corresponding states and state transitions as well as other relevant domain independent conditions. The formula Φ is a conjunctions of the individual formulas $\Phi_{A_0} \wedge (\bigwedge_{i=1}^{n} \Phi_{A_i}) \wedge \Phi_{independent}$ as seen previously in definition (4.3.36). It is satisfiable if a *finite model* $\mathcal{I}$ exists so that the formula Φ is entailed, i.e., iff $\mathcal{I}, \Phi_{A_0} \wedge (\bigwedge_{i=1}^{n} \Phi_{A_i}) \wedge \Phi_{independent} \models \Phi$.

Lemma 4.3.43 (**DPDL Formula Satisfiability**). *Given a target service E_0 w.r.t. a community of component services $E_1, \ldots, E_n$ and a DPDL formula Φ encoded as described in definition* (4.3.36)*. If Φ is satisfiable, there exists finite model $\mathcal{I}$ with $\mathcal{I}, \Phi_{A_0} \wedge (\bigwedge_{i=1}^{n} \Phi_{A_i}) \wedge \Phi_{independent} \models \Phi$ such that any such model represents a composition of E_0 w.r.t. $E_1, \ldots, E_n$.*

Proof. (sketch) A proof as sketch is referenced in [Berardi et al., 2005c]. □

Theorem 4.3.44 (**Service Composition Existence**). *The DPDL formula Φ is satisfiable iff there exists a composition of E_0 w.r.t. $E_1, \ldots, E_n$. Checking the existence of composition is EXPTIME-complete with a complexity lower bound recently proved in [Muscholl & Walukiewicz, 2007, Muscholl & Walukiewicz, 2008].*

□

Mealy Composition Synthesis

In order to realize a composition by synthesis, we reiterate the idea of a service orchestrator described in definition (4.3.22). We observe in theorem (4.3.44) that if Φ is *satisfiable* it admits a *model* which is the composition of the target service E_{target} (E_0); else if Φ is not satisfiable, no model exists. From a model an internal schema for the target service E_0 can be extracted which is a Mealy finite state machine (MFSM) composition of E_0 w.r.t. the community of component services $E_1, \ldots, E_n$ where E_0 acts as the orchestrator. The size of the model is at most exponential in the size of Φ which is a characteristic of the *small model property* described in definition (4.3.32).

Definition 4.3.45 (**Mealy Composition**). Given a *finite model* $\mathcal{I}_{fm}$ of the Kripke structure with $\mathcal{I}_{fm} = (\triangle^{\mathcal{I}_{fm}}, \left\{a^{\mathcal{I}_{fm}}\right\}_{a \in \Sigma}, \left\{P^{\mathcal{I}_{fm}}\right\}_{P \in \mathcal{P}})$ iff Φ is satisfiable and admits the model. A *Mealy composition* A_{Mealy} is the Mealy finite state machine (MFSM) of the Mealy internal schema of the target service with $A_{Mealy} = (\Sigma, \sigma_{Mealy}, S_{Mealy}, s^0_{Mealy}, \delta_{Mealy}, \omega_{Mealy}, F_{Mealy})$. The Mealy composition can be extracted from the model $\mathcal{I}_{fm}$ obtained by:

- $S_{Mealy} = \triangle^{\mathcal{I}_{fm}}$ specifying the set of possible states of the MFSM is derived from the domain model of the Kripke structure;
- $s^0_{Mealy} \in (s^0_0 \wedge \bigwedge_{i=1}^{n} s^0_i)^{\mathcal{I}_{fm}}$ specifying the initial state of the MFSM;
- $s' = \delta_{Mealy}(s, a)$ iff the pair $(s, s') \in a^{\mathcal{I}_{fm}}$ holds, i.e., the state and the successor state is defined in the action propositions of the model which specifies action a and the corresponding state transition;
- $\sigma_{Mealy} = \omega_{\mathcal{I}_{fm}}$ iff $(s, s') \in a^{\mathcal{I}_{fm}}$ and $\forall i \in \sigma_{Mealy}, s' \in \mathsf{moved}_i^{\mathcal{I}_{fm}} \;;\; j \notin \sigma_{Mealy}, s' \notin \mathsf{moved}_j^{\mathcal{I}_{fm}}$ specifying the state transitions via the Mealy output function stating which transitions are defined and those that are not;
- $F_{Mealy} = F_0^{\mathcal{I}_{fm}}$ specifying the final state of the MFSM.

□

Corollary 4.3.46 (**Mealy Composition Synthesis**). *Any finite model of the DPDL formula Φ denotes a synthesized Mealy composition of target service E_0 w.r.t. component services $E_1, \ldots, E_n$. The size of the Mealy composition is at most exponential in the size of the FSM external schemas $A_0, A_1, \ldots, A_n$ of the corresponding services respectively; which is a characteristic of the small model property of a DPDL formula as defined in definition* (4.3.32).

□

Algorithm 4.3.1 Algorithm for synthesizing Mealy composition

```
1: INPUT: A_0              /* FSM external schema of target service */
2:        A_1, ..., A_n    /* FSM external schema of community component services */
3:
4: OUTPUT: A_Mealy | nil   /* Algorithm returns a Mealy composition if one exist, else returns nil */
5:
6: BEGIN
7:   Φ := External_FSM_to_DPDL_Formula(A_0, A_1, ..., A_n);
8:   I_fm := DPDL_Tableau(Φ);
9: if (I_fm == nil) then
10:      return nil;
11: else
12:      A_Mealy := Extract_MFSM(I_fm);
13:      A_Mealy_min := Minimize_MFSM(A_Mealy);
14:      return A_Mealy_min;
15: end if
16: END
```

The pseudocode shown in algorithm listing **??** describes the synthesis of Mealy composition. Input to the algorithm is the FSM external schema A_0 for the target service and the FSM external schemas for the community component services. A DPDL formula Φ as described in definition (4.3.36) is built using the function External_FSM_to_DPDL_Formula. Taking into account the target service requested by the client, the available services of the community and the domain independent conditions, Φ encodes all real services (community services) and virtual service (target service) participating in the composition, characterizing which service in the community can take on a transition (moved) in correspondence with each transition of the target service so that the domain independent conditions are satisfied. The next step is to find out if a finite model (a Mealy composition) of Φ exists, this is done by using the DPDL_Tableau function which exploits tableau algorithms[38] [Giacomo & Massacci, 2000, Giacomo & Massacci, 1996] to check satisfiability of Φ. The function takes Φ as input argument and returns a finite model if Φ is satisfiable; otherwise the special placeholder nil is returned indicating no model exists. According to definition (4.3.45), the function Extract_MFSM transforms from the finite model $\mathcal{I}_{fm}$ into a MFSM internal schema of the target service A_{Mealy} by discarding from each state of $\mathcal{I}_{fm}$ the information about the current states of each component service and keeping in $\mathcal{I}_{fm}$ only the information about the final states or the state transitions. Such a model could contain redundant information such as unreachable or unnecessary states. The function Minimize_MFSM is used to perform minimization on the model. The minimized MFSM is returned as synthesized Mealy composition where each action of the target service is annotated with the service in the community that executes it. An implementation of the algorithm can be found in [Berardi et al., 2004a].

[38] A general account on tableau reasoning for optimized satisfiability checking can be found in [Baader et al., 2007, Baader & Sattler, 2001].

4.3.5 Characterization of Services Composition in the Situation Calculus

To shed light on the composition problem and the technique to check the existence of a composition described in section 4.3.4 from another perspective which attempts to characterize dynamic aspects of services in terms of actions and state changes by the effect that the actions bring about. The situation calculus can be used as a formalism to reason about actions representing service operations. In particular it can be used to build an action theory about the external schema of the target service, external schemas of the component services and the domain independent conditions in order to check satisfiability of the theory as described in section 4.3.4 about the problem and techniques of service composition. We focus on Reiter's situation calculus basic action theory [Reiter, 2001a, Levesque et al., 1998, Lin & Reiter, 1997, Pirri & Reiter, 1999] to tackle the known and discussed aspects of the service composition problem.

Behavior Description in the Situation Calculus

The external schema of a service as described in definition (4.3.5) can be expressed in a *situation calculus basic action theory* $\mathcal{D}$ in which each atomic action a is represented by a *named situation calculus action*. The DPDL atomic propositions can be expressed as situation calculus *fluents* whose trueness depends on the situation encoded in the parameter (situation variable) of the fluent. In this sense a state s in the previous encoding a DPDL proposition is analogous to a situation s in the situation calculus action theory and each DPDL proposition can be encoded as a situation calculus fluent whose trueness depends on the specific state of the finite state machine (FSM) external schema of a service described in the previous sections. Therefore a situation calculus basic action theory $\mathcal{D}$ consists of:

1. situation calculus axioms describing the *initial situation* S_0 where we have complete information in the initial situation;

2. one fluent F_a for each action a with the semantics that F_a is true in situation s iff it is possible to execute a in s w.r.t. a current state in the FSM;

3. one situation calculus *action precondition axiom* for each atomic action a in the form $\forall s.Poss(a, s) \equiv \Psi_a(s)$ where $\Psi_a(s))$ is a situation calculus formula with s as the only free variable where $Poss$ itself does not appear;

4. one situation calculus *successor state axiom* for each fluent F_a of the form $\forall a, s.F(do(a, s)) \equiv \Phi_F(a, s)$ where $\Phi_F(a, s)$ is a situation calculus formula with a and s as the only free variables;

5. a special devised fluent $Final$ which denotes the execution of a service can stop in the situation associated with the fluent. It corresponds to the specification of a final state in a DPDL proposition;

6. *unique name axioms* for the atomic actions and some domain independent axioms.

We observe the situation calculus *action precondition axioms* and *successor state axioms* together characterize the semantics of a corresponding state transition proposition over an executable action. We also notice that we have complete information in every situation on the state transitions defined over atomic actions in the action alphabet $a \in \Sigma$ because of action precondition axioms and successor state axioms.
A situation tree in the situation calculus [Reiter, 2001a] formed by executable actions, i.e., where the action precondition axiom for the respective action holds, directly relates to the external execution tree of the service where the nodes with a *final* state (interaction with the service can be terminated) are the situations in which the fluent $Final$ is true. The situation calculus basic action theory can be unfolded into a situation tree with each node denoting a situation s and each edge an executable atomic action a where $Poss(a, s)$ holds. The tree therefore characterizes actions that are possible to execute as a set of action trajectories from the root and its topology resembles that of the previously described external execution tree of a service external

schema.

Definition 4.3.47 (Situation Calculus Execution Tree). Given an FSM external execution tree $T(\mathcal{A}^{ext}(E))$ of a service as described in corollary (4.3.28), a corresponding situation calculus execution tree $T(\mathcal{D})$ w.r.t. a service community is formally defined inductively using a mapping function $map(\cdot)$ starting from the root node ε (where no action has been performed yet) to associate each successive node with a situation s and an action (or a set of actions if there is more than one action possible) that is executable from s. This approach directly maps out the execution of atomic actions of a service based on the FSM external schema representing the service as a tree of execution steps where the only defined possible actions are those actions whose *action precondition axioms* $\forall s.Poss(a, s)$ hold. We define a special placeholder fluent $undef$ for undefined situation and proceed with the construction of the situation tree:

- the root node ε is obtained from the situation calculus initial situation S_0 with $\varepsilon = map(S_0)$;
- the immediate successive node(s): $map(do(a, S_0)) = map(S_0) \cdot a$ iff $Poss(a, S_0) \wedge map(S_0) \neq undef$ holds;
- for all other nodes: $\forall s' = do(a, s) \wedge s \neq S_0 \rightarrow map(do(a, s)) = map(s) \cdot a$ iff $Poss(a, s) \wedge map(s) \neq undef$ holds;
- where action precondition axioms do not hold, assign the node $map(do(a, s)) = undef$ where $\neg Poss(a, s)$;
- mark nodes as final if $Poss(a, s) \wedge map(s) \neq undef \wedge Final(s)$ holds;

the part of the situation tree formed by executable actions is finitely representable.

□

Automatic Service Composition in the Situation Calculus

To recapitulate, the essence of composition in the *Roman Model* is the idea of delegating of atomic actions to a community of available services to execute and coordinate interactions between the client, target service and the component services using the concept of an orchestrator. In this sense the characterization of automatic composition and composition synthesis is also based on the notion of a target service E_0 and a set of component services $E_1, \ldots, E_n$. The idea is to represent which services are executed when a action of the target service is performed w.r.t. its external execution tree. A special situation calculus fluent $Step_i(a, s)$ is added with the subscript identifier i indicating the specific service which executes an action a in situation s. For the target service we formulate a situation calculus action theory $\mathcal{D}_0$ and for the component services $\mathcal{D}_1, \ldots, \mathcal{D}_n$. The union of these axioms represents the holistic basic action theory $\mathcal{D}_C \leftrightarrow \mathcal{D}_0 \cup (\bigcup_{i=1}^{n} \mathcal{D}_i)$ for the service composition problem.

Definition 4.3.48 (Situation Calculus Action Theory Encoding for Composition). Let $\mathcal{D}_0$ be the set of axioms constituting the situation calculus basic action theory for the target service and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ be the sets of axioms for the component services. The situation calculus action theory for the composition problem is the union of these axioms $\mathcal{D}_C$ with:

- $\mathcal{D}_0$ is axiomatized as described previously;
- the union $(\bigcup_{i=1}^{n} \mathcal{D}_i)$ with each set of axioms $\mathcal{D}_i$ corresponding to a component service in the community characterizes the action theory of each component service. $\mathcal{D}_i$ is obtained by:
  1. renaming each fluent F in $\mathcal{D}_i$ including the $Final$ axiom to F_i and $Final_i$ thus associating the fluents to the component service for identification;

2. renaming the set of action precondition axioms $Poss$ to $Poss_i$ for similar reason;
3. modifying the set of successor state axioms with:
   $\forall a, s.F_i(do(a, s)) \equiv (Step_i(a, s) \wedge \Phi_{F_i}(a, s)) \vee (\neg Step_i(a, s) \wedge F_i(s))$ augmenting the action theory with information on which service executes an action encoded with the fluents $Step_i$, i.e., either a component service executes action a by delegation and a successor situation is reached or the component service does not execute the action and the situation does not change[39];

- $\forall a, s.(Poss(a, s) \wedge \neg Final(s)) \supset \bigvee_{i=1}^{n} Step_i(a, s) \wedge Poss_i(a, s)$ this axiom belongs to the domain independent axioms and constraints the value of $Step_i$ specifying that at least one of the component service steps forward (is delegated an action and executes it) if an action is executable and the state is not final;
- $\forall s.Final(s) \supset \bigwedge_{i=1}^{n} Final_i(s)$ specifies if target service is in a final state, all the component services are too in a final state;

the mentioned axioms of initial situation, unique name axioms and domain independent axioms are implicit and must be present in each set of axioms for the services.

□

Example 4.3.49. We consider our guiding example of a stock investment scenario including a target service and two component services as shown in the figures 4.10, 4.11 and 4.12 of section 4.3.4. We recall that the set of atomic actions belongs to the action alphabet $\Sigma = \{a, t, l\}$ with "a" being a shorthand for request_up_to_date_stock_quote, "t" for request_stock_quote_history and "l" for list_and_chart_data for these services. On the basis of example (4.3.38) of the illustration of a DPDL encoding of our target service E_0, we show the axiomatization of the involved finite state machine of E_0 using the situation calculus formalism to encode axioms in $\mathcal{D}_0$:

$$\begin{array}{lcl}
\forall s.Poss(a, s) & \equiv & Final(s) \\
\forall s.Poss(t, s) & \equiv & Final(s) \\
\forall s.Poss(l, s) & \equiv & \neg Final(s) \\
\forall \alpha, s.Final(do(\alpha, s)) & \equiv & (\alpha = \exists act.Poss(act, s) \wedge \neg Final(s)) \vee \\
 & & (Final(s) \wedge \alpha \neq a \wedge \alpha \neq t)
\end{array}$$

For the component service E_1 due to parsimonious reason $\mathcal{D}_1$ axioms are shown briefly as follows:

$$\begin{array}{lcl}
\forall s.Poss_1(a, s) & \equiv & Final_1(s) \\
\forall s.Poss_1(t, s) & \equiv & \mathsf{false} \\
\forall s.Poss_1(l, s) & \equiv & \neg Final_1(s) \\
\forall \alpha, s.Final_1(do(\alpha, s)) & \equiv & (Step_1(\alpha, s) \wedge \alpha = \exists act.Poss(act, s) \wedge \neg Final_1(s)) \vee \\
 & & (\neg Step_1(\alpha, s) \wedge Final_1(s) \wedge \alpha \neq a)
\end{array}$$

and for E_2 the action theory axioms $\mathcal{D}_2$ are:

$$\begin{array}{lcl}
\forall s.Poss_2(a, s) & \equiv & \mathsf{false} \\
\forall s.Poss_2(t, s) & \equiv & Final_2(s) \\
\forall s.Poss_2(l, s) & \equiv & \neg Final_2(s) \\
\forall \alpha, s.Final_2(do(\alpha, s)) & \equiv & (Step_2(\alpha, s) \wedge \alpha = \exists act.Poss(act, s) \wedge \neg Final_2(s)) \vee \\
 & & (\neg Step_2(\alpha, s) \wedge Final_2(s) \wedge \alpha \neq a)
\end{array}$$

[39] Therefore the modified successor state axioms do not only depend on the previous situation but also on the trueness of $Step_i$. We observe that this proposition resembles the predicates moved and ¬moved in the previous DPDL proposition description.

Initial situation is encoded as $\mathcal{D}_{init}$:

$$Final(S_0)$$

Domain independent conditions are encoded in $\mathcal{D}_{independent}$ as follows:

$$\begin{array}{lll} \forall a, s.(Poss(a,s) \wedge \neg Final(s)) & \supset & (Step_1(a,s) \wedge Poss_1(a,s)) \vee \\ & & (Step_2(a,s) \wedge Poss_2(a,s)) \\ \forall s.Final_0(s) & \supset & Final_1(s) \wedge Final_2(s) \end{array}$$

With this situation calculus characterization of the services, the problem of checking for existence of a composition can be reduced to checking satisfiability of the situation calculus action theory $\mathcal{D}_C$.

Theorem 4.3.50 (**Situation Calculus Action Theory Satisfiability**). *Let $\mathcal{D}_C$ be the union of situation calculus basic action theories axiomatizing the target service, a community of component services of size n as described in definition* (4.3.48) *and the domain independent axioms. The theory $\mathcal{D}_C$ is satisfiable if it admits a model and:*

$$\mathcal{D}_{init} \cup \mathcal{D}_{independent} \cup \mathcal{D}_0 \cup (\bigcup_{i=1}^{n} \mathcal{D}_i) \models \mathcal{D}_C$$

□

Theorem 4.3.51 (**Situation Calculus Action Theory Composition**). *Let $\mathcal{D}_C$ be the situation calculus basic action theory axiomatization representing a target service E_0 and a community of component services $E_1, \ldots, E_n$ respectively. A composition of services exists if $\mathcal{D}_C$ is satisfiable and admits a finite model. By the constraints posed to the interpretation of $Step_i(a,s)$ fluents for each component service E_i, the admitted model of the basic action theory is a composition in the sense that an internal execution tree of the target service E_0 can be constructed from the action trajectory containing for each action a and situation s the delegable services where $Step_i(a,s)$ holds and therefore E_0 can be composed using $E_1, \ldots, E_n$.*

□

5 Action Theories for Service Composition

We turn our attention to a process scenario which serves as a guiding example through this chapter. The requirement of integration of existing process oriented enterprise services should meet the goal of synthesizing a process scenario for registration of a business with a public administration. The existing enterprise services are not semantically interoperable with intelligent agents because there is no logical axiomatization of their behavior to model preconditions and effects of these services. The dynamical aspects as well as constraints of the enterprise services and their underlying processes must be axiomatized in an appropriate way.

We can make use of the theoretical foundations described in chapter 4, especially using the action theory of the situation calculus to model the constraints of the services as action precondition and successor state axioms. The service composition problem can be reduced to a theorem proving task of the axiomatized application domain, in our case a guiding process scenario of business registration at a government public administrative organization which is further described in this chapter. To make the problem approachable to focus on the service synthesis aspect, we do ignore the aspects of dataflow of process oriented enterprise services under the assumption that dataflow is appropriately modeled and that the required types of the input parameters and outputs are captured semantically in enterprise service ontology that exists internally.

In fact deriving logical inferential information of process order and obtaining a correct execution sequence for the services to solve the composition problem is more challenging task to us because no single enterprise service can satisfy the functional requirements of the process scenario alone. The situation calculus appeals to us strongly as a powerful tool to implement the dynamical process scenario for describing enterprise service composition. The situation calculus action theory proves to be an ideal formalism for functional synthesis because we can abstract away from dataflow and concentrate on the dynamical aspects of specification, i.e., answering the question of how to combine components to get resulting functionality in due consideration of the existing process and service constraints. The issue of semantic service matching is not a central requirement to our process integration scenario because the advertisements and documentations of service capability exist already and enterprise services are well bundled according to their application domain objectives such that it is rather intuitive to understand their designation. Furthermore the total number is finite since we are not dealing with volatile service repositories where service configuration changes often. The fact that we are subject to composition synthesis of service functionalities using a set of existing services which are registered in a repository has motivated us to tackle composition directly. For the process scenario, we have selected to integrate a finite set of services from the CRM bundle[1], business service application domain and banking application domain bundles to get the partial functionality that we desire. What these services lack is detailed descriptions of their *application semantic* such as formal definitions of invocation semantics, preconditions which enable invocation and postconditions of execution. There is also little apparent about application domain constraints when combining the existing services. Fortunately we can obtain partial information for our axiomatization effort from the status and action management runtime information as described previously in chapter 2, in order to figure out rather cursorily the required service semantics.

[1] An assorted CRM-related group of enterprise services are bundled together in an enterprise service module for efficient deployment.

5.1 Guiding Process Scenario

We start with a guiding process scenario that represents a business process for business registration at governmental administrative organization. Motivation comes from business requirement nowadays to automate public sector administrative processes.

5.1.1 Supporting Automated Business Registration with Enterprise Services

Figure 5.1 illustrates this business process scenario in the business process modeling notation (BPMN). It is a kind notational diagram to visualize process components and process activities and as such it does not represent execution syntax but it can be transformed to a specific process execution language using appropriate process modeling tools.

This business process consists of four main phases with each one contained in a labeled phase box as depicted in the figure:

i) **creation phase**: the main activity is to receive incoming cases of application for business registration. A citizen intends to start a business and a business record must be created in the registries of the governmental administrative department for commerce. Assume that there exists installation of SAP enterprise services of the customer relationship management (CRM) bundle which has the following functionality

   a) check whether a case of registration is open in the CRM system;

   b) check whether a citizen exists in the CRM system;

   c) creation of record for a new citizen if she is not yet covered by the system;

   d) for each citizen check whether bank account details exist so that governmental agency can bill the citizen for using the registration service[2];

   e) if no bank detail of a citizen exists currently, a CRM process activity is initialized to create a corresponding record in the CRM system.

ii) **pre-check phase**: the purpose of pre-checking is to scrutinize the application case to examine essential benchmarks of an application case approximately, for instance, clearance of the applicant's taxation register to review lawfulness, checking business location in the application and checking applicant's record of conduct clearance with police department, etc. A parallel **denial phase** can be chosen to deny application case if pre-check criteria and conditions are not complied with. The denial phase consists of the activities to send denial and archive the denied case on the registration application system. Process activities of the pre-check phase are:

   a) check applicant's business location to evaluate availability of a requested location. If such business location is already occupied, reject the application;

   b) examine the lawfulness of the nature of the requested business registration, for instance, checking whether a conflict situation exists due to antitrust accusation, distortion of competition, violation of existing patents or violation of copyright, etc. Each business executive must also be scrutinized for past record of conduct.

[2] It is assumed that beside the CRM aided process phase in the current process scenario, other registration procedures of a governmental agency must be subject to other process activities under supervision of financial administration to evaluate credit worthiness and solvency of a citizen as well as lawfulness of her capital financing must be scrutinized. These related process activities must be performed for security reasons not on an enterprise system but manually by government agents. Therefore no automated support via business process activities for these procedures is required.

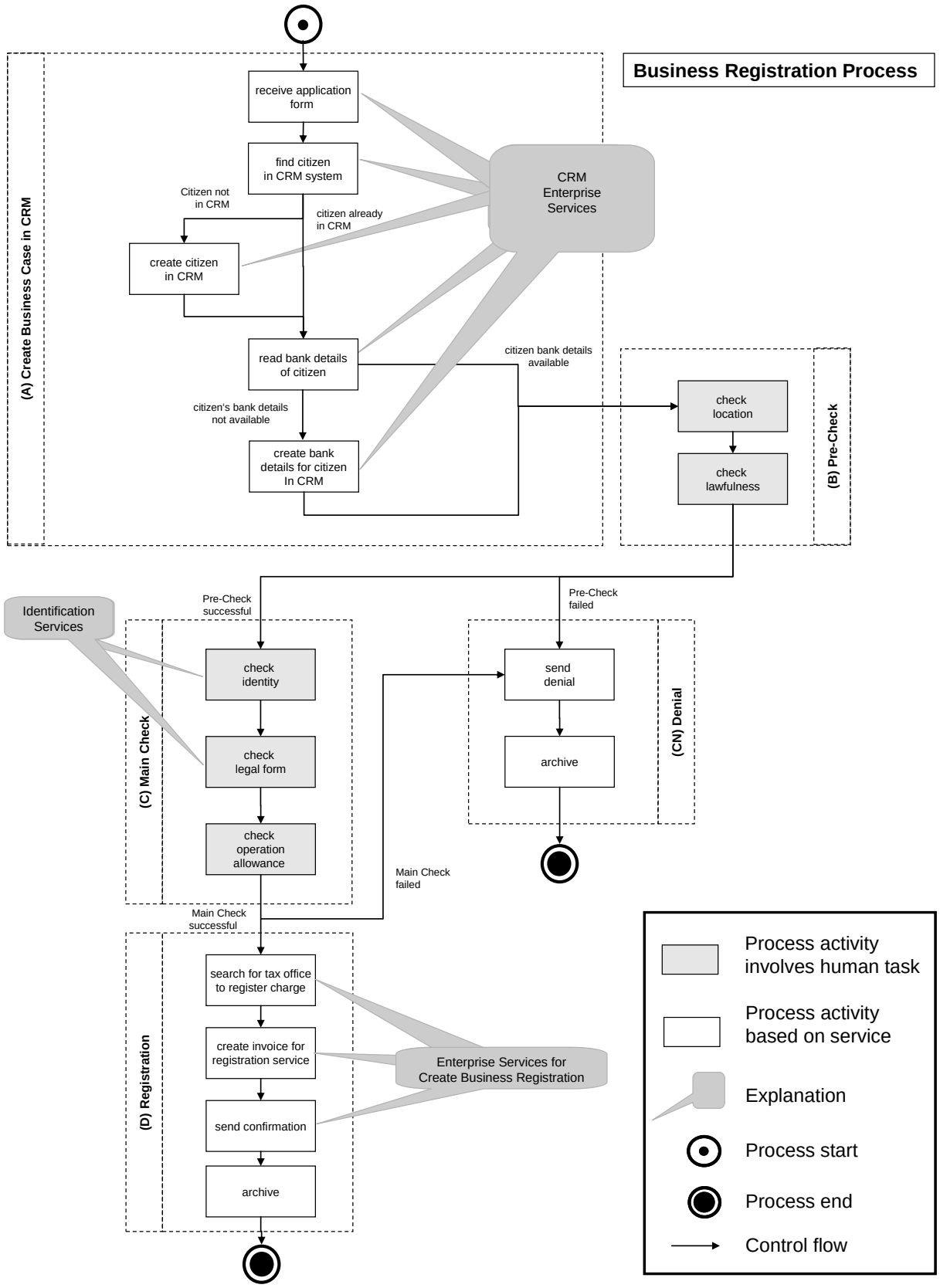

Figure 5.1: Process scenario for business registration at governmental administration

iii) **main check phase**: the purpose of main check phase is to investigate detailed profile of applicants as well as legal form of the business. If conditions in this phase are not complied with, the denial phase is entered. Activities of the main check phase are:

   a) an agent uses identification services to check identification information of the business executives, business holders. Due to protection of privacy, these services are not enterprise services and are accessed only within the governmental administrative network;

   b) the legal form of the business must be checked according to scope of business, operational activities, business areas, import and export regulations, etc. Only governmental agent can access legal internal data for these activities and it is not modeled with SAP processes;

   c) other legal issues must be resolved on the operational activities of a business before license of entitlement to entrepreneurship can be granted.

iv) **registration phase**: this last phase of the business process establish the persistent business registration of a case in the following activities:

   a) search for the tax office responsible for applicant's business location and connect through record registries with it. If there is no appropriate registry for the case, enterprise service on the central registry is used in order to establish profile for collecting registration service charges;

   b) create invoice to bill business registration service by charging the applying citizen;

   c) if registration process is successful, send applicant confirmation and archive the case for possible review later.

5.1.2 Composite Process Functional Requirements

Our scenario is represented in the mentioned business phases which integrates functionalities across several enterprise services bundles, for instance, the customer records management services and banking record services of the CRM bundle and business registration of the government services bundle.

Requirements of process phase (A) in figure 5.1 consist of a holistic view of the functionalities:

a) capability to receive application form and verify well-formedness and validity of key value fields in the form;

b) ability to correctly extract the entire set of relevant data from the form for further processing;

c) capable of checking syntax correctness and formatting of the extracted data;

d) storing this information in an internal database on the SAP ERP system;

e) ability to access CRM database to determine whether the applying citizen[3] has already been captured in CRM system;

f) capable of implicitly deriving existing applicant profile and business history from the CRM system;

g) capable of visualizing this profiling information when required for internal scrutiny;

h) ability to create an applicant record and initialize it appropriately;

i) ability to connect to enterprise services of the banking industry bundle to examine required business accounting information for billing purpose;

[3]For brevity the applying citizen entity will be referred to as *applicant* hereafter.

j) capable of creating new business accounting profile for applicants;

k) ability to search for information of an applicant's banking account payable and receivable profile;

l) if necessary, ability to modify an applicant's bank details such as bank key, additional account number, solvency ranking, etc.;

m) ability to synchronize bank details with master data of application process component.

Since process phases (B) and (C) in figure 5.1 are processed predominantly manually and must not be realized via enterprise service due to previously mentioned reasons, we can assume that their required functionalities can be realized accordingly and view them as some sort of black-box process components to consolidate the control flow further. In this sense these phases can be viewed as individual consolidated functional unit that, when inputs are provided appropriately in the form of application data and documents, they will execute and reach their phase goals without having the enterprise services participating. We should see that these phases can be modeled subsequently as single action respectively which when executed by an agent just interleaved with other operators and controller actions in the composition, i.e., abstracting away from the constituting process activities or involved services therein.

Process phase (D) in figure 5.1 represents the business activities that support registration process at governmental agency and requires these functionalities:

a) interoperable with enterprise services of the financial bundle at taxation agency;

b) capable of accessing taxation record of individual applicant;

c) ability to extract from existing taxation records relevant fields for registration, for instance name of taxation authority responsible for a district, applicant's taxation number, types of taxation due and whether certain relief conditions apply;

d) ability to transfer the taxation record information to an applicant's case profile and store it persistently;

e) capable of updating taxation profiles in database of the financial module to reflect new conditions after setting up business;

f) ability to register and persist a business opening and determine the type of business in the applicant's profile;

g) capable of determining the correct amount of due taxes for the specific business type;

h) capable of creating invoice and with appropriate amount of service charges accordingly to bill the customer;

i) capable of archiving processed registration for future review and auditing.

The process goal can be defined therefore as: *adapting a subset of functionalities of several process bundles to satisfy the need to register business semi-automatically by coordinating external discrete governmental processes or services*. The plethora of diverse functional requirements dictate a necessary combination of existing functionalities of enterprise services to produce the overall required functionality. Since no single enterprise process can solely satisfy our process goal, we resort to synthesizing selected enterprise services and coordinating them to achieve the goal.

5.1.3 Selected Enterprise Services

For synthesis purpose a set of selected enterprise services across several enterprise services bundles are described in the following sections. We describe definition, technical data, business usage context and known constraints for each enterprise service. We will explicitly specify and describe these selected services[4] where for simplicity exceptions are provisionally trimmed in the current scenario. Due to often large enterprise service operations defined in enterprise service interfaces which require long list of input parameters, we have chosen to mask out a series of insignificant and optional input parameters that do not directly influence correct execution of the service operations.

Process Phase A

The selected services satisfy the current functional requirements partially. The enterprise service for technical processing and capturing application data is described in table 5.1.

Enterprise service which handles search of applicant's information according to the extracted application profile in CRM is carried out by the service described in table 5.2.

If there is no corresponding applicant profile in the CRM system, the enterprise service described in table 5.3 is responsible for creating it.

After CRM system has been updated, the enterprise service described in table 5.4 attempts to read bank details of the applicant if it exists.

For each applicant who has not deposited valid bank details in the CRM system, the enterprise service described in table 5.5 is used to create and initialize a corresponding bank detail profile.

Process Phase CN

Since phases (B) and (C) are not supported through public enterprise services, in phase (CN) denied applicants can be notified using the enterprise service described in table 5.6.

For archival of denied cases, the same enterprise service can be used as in final archival of processed application cases.

Process Phase D

Taxation matters are handled by the service described in table 5.7 where it obtains input from the process operation results of phase (C), assuming the same business application business object is continuously used in the internal business dataflow.

After a business registration application is processed and corresponding data is persisted, invoice can be created to bill the applicant using the service described in table 5.8.

[4] In the following tables, 'MEP' is a shorthand for message exchange pattern and the label 'Multi-operational' describes whether a service contains more than one operation defined in its service interface.

Service Aspects	Description
Definition	
	Accepting submitted application form electronically and extracting predefined attributes with handling of validation of captured data
Technical Data	
Technical Name	InboundNWSUITE_RDM_ProcessBRApplDoc_In_Sync
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	synchronous
Change/Update Behavior	ACID transactional
Idempotency	N.A.
Multi-operational	N.A.
Bundle Association	RecordDocumentManagementBundle
Parameters & Return Object	
Inputs	BusiRegEDocument, BusiRegType, BusiRegLocation, BusiRegIndustry
Output	PublicRecordDocument
Business Context	
	This inbound service extracts application form and extracts information related to a specific predefined business goal. It validates data related to specific business scenario of business application and registration.
Related Business Objects	
	PublicRecordDocument, BusiRegEDocument, BusiRegActor
Constraints	
	• Process does not support concurrent extraction and validation of application; • precondition of the operation is that there exists a submitted electronic application for registration; • postcondition is that a new application case in form of the associated business object is updated in the record and document management backend system; • erroneous application must be handled by different service.

Table 5.1: Enterprise service: Process application of inbound business registration

Service Aspects	Description
Definition	
	From the extracted applicant's profile information, this service searches the backend CRM system to determine whether the input profile information already exists. If is exists, the dataflow is relayed; otherwise CRM data is not forwarded.
Technical Data	
Technical Name	InboundNWSUITE_CRM_SearchByIDBRAppl_In_Sync
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	synchronous
Change/Update Behavior	N.A.
Idempotency	idempotent
Multi-operational	N.A.
Bundle Association	CRMBundle
Parameters & Return Object	
Inputs	BusiRegActorID
Output	BusiRegCustomer
Business Context	
	An agent or an official at a business registration administration can use this service to examine whether an applicant's profile is captured.
Related Business Objects	
	PublicRecordDocument, BusiRegCustomer
Constraints	
	• precondition is that a new application case is correctly capture so that the CRM system can access the document and record management backend system to get information of the case and use the application id to search in the CRM system; • postcondition is that an agent has obtained the knowledge that either an applicant's profile exists in CRM already or it does not; • invariance is that if a record already exists in CRM, its state and internal data structure is not changed.

Table 5.2: Enterprise service: Search profile of applicants in CRM

Service Aspects	Description
Definition	
	This service takes the extracted applicant's profile information as input and create a corresponding applicant in the backend CRM system. After creation values are initialized and dataflow object can be forwarded to subsequent process component.
Technical Data	
Technical Name	InboundNWSUITE_CRM_CreateInitBRAppl_In_Sync
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	synchronous
Change/Update Behavior	ACID transactional
Idempotency	N.A.
Multi-operational	N.A.
Bundle Association	CRMBundle
Parameters & Return Object	
Inputs	BusiRegActorID, *automated and human input*
Output	BusiRegCustomer
Business Context	
	Applicant's profile data is essential for the registration process. It contains the application actor identifier to identify an applicant uniquely. Moreover, it contains the actor id which is used to initiate certain process internal tasks. Further information of the applicant such as taxation number and bank information can be attached to an applicant's profile. This operation creates a record for the applicant's profile persistently within the CRM backend application system.
Related Business Objects	
	BusiRegCustomer, BusiRegCustomerID, BusiRegActorID, BusiRegCustomerBankInfo, BusiRegCustomerTaxID
Constraints	
	• The service operates only on general, insensitive customer data; • no duplicated profile will be created in the backend; • once an applicant's profile has been created, this operation will not be able to delete it, instead when registration process has finished, it can be transformed to the dormant state; • precondition is that no identical applicant profile exists in CRM; • postcondition is that a profile corresponding to the applicant is created and accessible in CRM.

Table 5.3: Enterprise service: Create and initialize CRM applicant profile

Service Aspects	Description
Definition	
	This service attempts to access applicant's bank detail in financial accounting module's backend database and returns a public view of the applicant's bank and financial information such as bank institute, bank account number, clearance number, taxation number, IBAN/BIC codes, etc.
Technical Data	
Technical Name	InboundNWSUITE_FI_SearchViewBankInfoByCRMID_In_Sync
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	synchronous
Change/Update Behavior	N.A.
Idempotency	idempotent
Multi-operational	N.A.
Bundle Association	FinancialAccountingBundle
Parameters & Return Object	
Inputs	BusiRegCustomerID
Output	BusiRegCustomerBankInfo
Business Context	
	An agent or an official at a business registration administration can use this service to examine whether an applicant has deposited valid bank account information.
Related Business Objects	
	BusiRegCustomerID, BusiRegCustomerAccount, BusiRegCustomer-AccountID, BusiRegCustomerAccountHolder
Constraints	
	• precondition is that there is an application case associated with an applicant in the CRM system and the applicant's id can be used to search for her bank account information in the backend financial accounting system; • postcondition is that an agent has gained knowledge to know either an applicant's back account information is captured or not; • invariance is that if a set of bank detail for the applicant already exists in the financial accounting system, its state and internal data structure is not changed.

Table 5.4: Enterprise service: Read bank information in CRM

Service Aspects	Description
Definition	
	This enterprise service attempts to create a new profile of applicant's bank detail in the financial accounting module's backend database and link this profile with a record in CRM backend for the applicant's registration case.
Technical Data	
Technical Name	InboundNWSUITE_FI_CreateInitBankInfoBRAppl_In_Sync
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	synchronous
Change/Update Behavior	ACID transactional
Idempotency	N.A.
Multi-operational	N.A.
Bundle Association	FinancialAccountingBundle
Parameters & Return Object	
Inputs	BusiRegCustomerID, *automated and human input*
Output	BusiRegCustomerBankInfo
Business Context	
	Applicant's bank account information is essential for the registration process. According to this information, taxation issues are resolved and the registration service is charged. This operation creates a record for the applicant's bank account information persistently in the backend financial accounting application system.
Related Business Objects	
	BusiRegCustomer, BusiRegCustomerBankInfo
Constraints	
	• As usual this service operates on public accessible view of insensitive bank account information of the applicant; • no duplicated bank detail will be created in the backend; • precondition is that there is an application case (business registration or other chargable services) with associated applicant created and stored in the CRM system so that the created bank detail is associated with the applicant by applicant's id; • postcondition is that an applicant's bank detail is created and accessible in the bankend financial accounting system.

Table 5.5: Enterprise service: Create bank information of applicant in CRM

Service Aspects	Description
Definition	
	This enterprise service sends a denial to reject an applicant's registration.
Technical Data	
Technical Name	InboundNWSUITE_CRM_RejectBRApplByCRMID_In_Sync
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	synchronous
Change/Update Behavior	ACID transactional
Idempotency	N.A.
Multi-operational	N.A.
Bundle Association	CRMBundle
Parameters & Return Object	
Inputs	BusiRegCustomerID, BusiRegEDocument
Output	*denial in paper and electronic form*
Business Context	
	An agent is informed of denial of a specific application case and sends a denial in both paper and electronic form to an applicant.
Related Business Objects	
	BusiRegEDocument, PublicRecordDocument, BusiRegCustomer
Constraints	
	• precondition of this operation is that there exists a rejection of the specified business registration application; • postcondition is that agent has notified the applicant that her case is rejected and closed. • invariant condition is that after the notification, the CRM and FI systems will not change the stored applicant's profile and bank detail and that the application case persists in the backend record and document management system.

Table 5.6: Enterprise service: Process application denial of inbound business registration

Service Aspects	Description
Definition	
	This service searches for the applicant's taxation register to declare registration service charges.
Technical Data	
Technical Name	InboundNWSUITE_BRPA_SearchByIDBRApplTaxRA_In_Sync
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	synchronous
Change/Update Behavior	ACID transactional
Idempotency	N.A.
Multi-operational	N.A.
Bundle Association	BusinessRegistrationPublicAdminBundle
Parameters & Return Object	
Inputs	BusiRegCustomerID, BusiRegEDocument, BusiRegCustomerBank-Info
Output	BusiRegCustomerTaxID
Business Context	
	An agent or tax department official can use this service to levy a tax on the charges of business registration service performed.
Related Business Objects	
	BusiRegCustomer, BusiRegEDocument, BusiRegCustomerBankInfo, BusiRegCustomerTaxID
Constraints	
	• precondition is that there exists an applicant's profile in CRM and her bank detail in FI so that one can search the applicant's taxation register to record the service charge; • postcondition is that a taxation register is located and the bank clearance is successful for charging the registration service and taxation department has recorded the transaction; • business registration charge is taxed only once and that the process must guarantee this property.

Table 5.7: Enterprise service: Search taxation register for charging service

Service Aspects	Description
Definition	
	This enterprise service initiates billing the applicant and outputs an invoice for the performed registration service.
Technical Data	
Technical Name	InboundNWSUITE_BRPA_BillByIDBRApplInvoicing_In_Sync
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	synchronous
Change/Update Behavior	ACID transactional
Idempotency	N.A.
Multi-operational	N.A.
Bundle Association	BusinessRegistrationPublicAdminBundle
Parameters & Return Object	
Inputs	BusiRegCustomerID, BusiRegCustomerBankInfo, BusiRegCustomer-TaxID
Output	*electronic and printed invoice*
Business Context	
	An agent or registration official can use this service to bill and send an invoice for having perform a business registration service on behalf of an applicant.
Related Business Objects	
	BusiRegCustomer, BusiRegCustomerBankInfo, BusiRegCustomer-TaxID, PublicRecordDocument
Constraints	
	• precondition is that there exists an applicant's profile in CRM and her bank detail is in FI so that an agent can record billing information for the applicant and initiate external payment process to debit an applicant's account; • postcondition is that an external payment process to debit an applicant's account is initiated and that the transaction is recorded in the applicant's profile as history in CRM as well as in the specific registration record; • business registration service is billed only once and that the process must guarantee this property.

Table 5.8: Enterprise service: Charge registration service and send invoice

Simultaneously government agency can choose to send registration confirmation to the applicant using service described in table 5.9.

Subsequently, successfully registered business is contained in the internal dataflow which can be permanently or temporarily archived using service referred in table 5.10.

5.2 Action Theory Applied

In the spirit of [McIlraith, 1999, Giacomo & Sardina, 2009] and based on the solid foundations on the basic action theory of the situation calculus described in section 4.2 of chapter 4, we attempt to derive a correct axiomatization of our process domain.

5.2.1 Preliminary Action Theory Axiomatization

The required complex functionalities can be synthesized by resorting to *model-based programming* which comprises reusable high level programs that capture sufficient procedure knowledge of how tasks or process activities can be accomplished without specifying all system specific details. A *model-based program* consists of (i) a *model* which is an accurate and integrated representation in terms of formal appropriate axiomatization of the abstract problem domain. The model explicates behavior of the process components in the domain that are being programmed, the operators and controller actions that affects the model and states of the process components; (ii) a *program* which describes the model with high level procedures such as using the languages GOLOG/ConGolog[5] to realize the model using the defined operators or controller actions and states of the domain. We adopt the proven approach of representing process activities, supported with operations of existing enterprise services, as these operators and controller actions in the high level generic procedures are also reusable.

[5]In the following text we include the term ConGolog also when we mention GOLOG.

Service Aspects	Description
Definition	
	This service performs confirmation and sends notification.
Technical Data	
Technical Name	InboundNWSUITE_BRPA_ConfirmByIDBRAppl_In_Sync
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	synchronous
Change/Update Behavior	ACID transactional
Idempotency	N.A.
Multi-operational	N.A.
Bundle Association	BusinessRegistrationPublicAdminBundle
Parameters & Return Object	
Inputs	BusiRegCustomerID
Output	*electronic mail and official printed confirmation*
Business Context	
	An agent uses this service to send notification to an applicant to confirm the business registration process is completed successfully.
Related Business Objects	
	BusiRegCustomer
Constraints	
	• precondition is that there exists an applicant's profile in CRM so that an agent can look up contact information; • postcondition is that a confirmation is sent by an agent to inform the applicant and that the applicant's profile and registration record is marked as notified in the backend CRM system; • invariant condition is that the finished registration transaction is not changed; • multiple confirmations are not desirable but will not have influence on the complete business registration process.

Table 5.9: Enterprise service: Process registration send confirmation

Service Aspects	Description
Definition	
	This enterprise service transforms the finished business registration process and relevant documents to dormant on the enterprise document and record management system.
Technical Data	
Technical Name	InboundNWSUITE_RDM_DormantApplDocByIDBRAppl_In_Async
Namespace	http://sap.com/xi/NW/APPL/Global
MEP	inbound
Mode	asynchronous
Change/Update Behavior	N.A.
Idempotency	N.A.
Multi-operational	N.A.
Bundle Association	RecordDocumentManagementBundle
Parameters & Return Object	
Inputs	BusiRegEDocument
Output	N.A. (*application acknowledgement*)
Business Context	
	An agent uses this service to archive a processed business registration case which means that the record is transferred to a dormant state in order to store it permanently for bookkeeping purpose. If necessary it can be retrieved later for review or scrutiny.
Related Business Objects	
	BusiRegEDocument, PublicRecordDocument
Constraints	
	• precondition is that there exists an applicant's profile and a certain associated registration record that is finished in CRM so that an agent can transferred it into dormant state; • postcondition is that the record is dormant but not deleted; • invariant condition is that the applicant's relevant bank account information or taxation information is not changed; • the characteristical interaction mode is asynchronous. An agent lets the document management system proceed with the archival while it moves on and is subsequently notified when archival is finished.

Table 5.10: Enterprise service: Process dormant registration application for archival

Service Operations	**Complex Actions in $\mathcal{L}_{sitcalc}$**
InboundNWSUITE_RDM_ProcessBRApplDoc_In_Sync [table 5.1]	**RecvRegApplDoc**(RegApplDoc,RegType,RegLoc,RegIndust)
InboundNWSUITE_CRM_SearchByIDBRAppl_In_Sync [table 5.2]	**SearchApplProfile**(BusiRegActorID)
InboundNWSUITE_CRM_CreateInitBRAppl_In_Sync [table 5.3]	**CreateApplProfile**(BusiRegActorID)
InboundNWSUITE_FI_SearchViewBankInfoByCRMID_In_Sync [table 5.4]	**SearchApplBankInfo**(BusiRegCustomerID)
InboundNWSUITE_FI_CreateInitBankInfoBRAppl_In_Sync [table 5.5]	**CreateApplBankInfo**(BusiRegCustomerID)
InboundNWSUITE_CRM_RejectBRApplByCRMID_In_Sync [table 5.6]	**RejBusiReg**(BusiRegCustomerID,RegApplDoc)
InboundNWSUITE_BRPA_SearchByIDBRApplTaxRA_In_Sync [table 5.7]	**RecApplTaxInfo**(BusiRegCustomerID,RegApplDoc,CustBankInfo)
InboundNWSUITE_BRPA_BillByIDBRApplInvoicing_In_Sync [table 5.8]	**BillAppl**(BusiRegCustomerID,CustBankInfo,CustTaxID)
InboundNWSUITE_BRPA_ConfirmByIDBRAppl_In_Sync [table 5.9]	**ConfirmReg**(BusiRegCustomerID)
InboundNWSUITE_RDM_DormantApplDocByIDBRAppl_In_Async [table 5.10]	**ArchiveReg**(RegApplDoc)

Table 5.11: Mapping of service operations to corresponding complex actions in the situation calculus

The fact that all our selected enterprise services are not multi-operational, i.e., they expose only one operation for each service interface is favorable for axiomatization since each operation corresponds to one service which can be modeled as one generic procedure consisting of world-altering complex action in the situation calculus domain action theory. Otherwise we would have used multiple complex actions to represent all operations within the same service; as a result necessarily complicating our model. Consequently we have taken this fact as an advantage in our model that each service possesses exactly one world-altering complex action.

Each complex action in our domain action theory encapsulates procedural knowledge in $\mathcal{L}_{sitcalc}$ and comprises a set of primitive actions in the situation calculus that when performed exhibit the behavior of the procedural complex action. The technical name of each service is therefore taken and mapped to a corresponding complex action with input parameters in the argument list. This mapping of actions to services are listed in table 5.11. Naturally the complex actions will be defined as generic procedures and further broken down into their situation calculus primitive actions so that we can build an action theory that axiomatizes our scenario.

5.2.2 Building Domain Model

We recall that an axiomatization in the situation calculus language $\mathcal{L}_{sitcalc}$ consists of the following axioms:

- foundational axioms of the situation calculus Σ
- set of unique names axioms for actions and domain closure axioms $\mathcal{D}_{una}$,
- set of axioms $\mathcal{D}_{S_0}$ describing the initial situation S_0,
- action precondition axioms, one for each primitive action $\mathcal{D}_{ap}$,
- successor state axioms, one for each fluent defined $\mathcal{D}_{ss}$.

We define the essential primitive actions that are necessary to constitute the procedural complex actions, with the primitive actions at our disposal a set of state constraints can be derived to characterize the business registration process scenario in a formal way.

Primitive Actions

For brevity the parameters taken by primitive actions are abbreviated and mentioned when necessary.

- $receive(rd) \rightarrow$ Receive an applicant's document $RegApplDoc$ into system buffer where rd is an abbreviated form.
- $return(rd) \rightarrow$ Return and ignore a business registration application document rd.
- $check\_syntax(rd) \rightarrow$ Check well-formedness of syntax and data in the business registration application document.
- $validate\_rd(rd) \rightarrow$ Validate the data in the business registration application document.
- $create\_rdbo(rd) \rightarrow$ Create an instance of SAP Business Object (BO) in system.
- $store\_rdbo \rightarrow$ Persistently store the registration application document BO instance in backend database.
- $retrieve\_rdbo(id) \rightarrow$ Retrieve an applicant's registration application case using an id.

- $login\_crm$ → Login to CRM (customer relationship management) application module.
- $access\_crm$ → Initialize to access information on the CRM module.
- $login\_fi$ → Login to FI (financial accounting) application module.
- $access\_fi$ → Initialize to access information on FI module.
- $lookup\_cusprof(apid)$ → Lookup an applicant's profile using an customer (applicant's) id.
- $create\_cusprofbo(apid)$ → Create an applicant profile BO to contain profile information identified with applicant' id $apid$ in CRM.
- $add\_cusprofbo(val)$ → Add a value to the applicant profile BO.
- $store\_cusprofbo$ → Persistently store the applicant profile BO instance in backend database.
- $display\_cusprof(apid)$ → Display information in a specific applicant's profile by id.
- $lookup\_binfo(apid)$ → Lookup an applicant's bank account information by id.
- $display\_binfo(apid)$ → Display bank detail of applicant by id.
- $create\_binfobo(apid)$ → Create a BO to contain new bank account information in FI associated with an applicant identified with $apid$.
- $add\_binfobo(val)$ → Append a value to bank detail BO.
- $store\_binfobo$ → Persistently store the bank detail BO instance in backend database.
- $deny$ → Reject registration.
- $archive(rd)$ → Make a business registration application dormant and archive it.
- $read\_tax(apid)$ → Read tax information by id.
- $record\_tx(taxid)$ → Record tax to levy in current charge.
- $bill(apid)$ → Charge an applicant for service.
- $send\_invoice(apid)$ → Send out invoice to applicant by id.
- $notify$ → Send confirmation notification to applicant.

Assuming that all these primitive actions have their unique name axioms correctly specified which for parsimonious reason is not shown here. We recall that fluents are dependent on situation to take either true or false value with the general form $F(s)$ or $F(\vec{x}, s)$.

Fluents

- $exist\_rd(s)$ → in situation s there exists a business registration application,
- $received(rd, s)$ → in situation s the business registration application document rd is received in the system buffer,
- $syntax\_ok(rd, s)$ → in situation s the syntax is correct and compliant,
- $is\_valid(rd, s)$ → in situation s the data contained in rd is valid and checked,

- $has\_rdbo(rd, s)$ $\rightarrow$ in situation s the registration application document BO identified with rd exists in backend,
- $stored(bk, s)$ $\rightarrow$ in situation s the a backend bk has performed a store operation,
- $accessible(bk, s)$ $\rightarrow$ in situation s the backend named with variable bk is accessible,
- $logged\_on(bk, s)$ $\rightarrow$ in situation s agent is logged on the backend named with variable bk,
- $has\_cusprofbo(apid, s)$ $\rightarrow$ in situation s the applicant's profile BO identified with $apid$ exists in backend,
- $has\_binfobo(apid, s)$ $\rightarrow$ in situation s the applicant's bank detail BO identified with $apid$ exists in backend,
- $has\_data(bk, s)$ $\rightarrow$ in situation s the named backend contains data,
- $writable(bk, s)$ $\rightarrow$ in situation s the agent can write/update date on the named backend,
- $rejectable(s)$ $\rightarrow$ in situation s a registration application can be rejected,
- $is\_dormant(rd, s)$ $\rightarrow$ in situation s the business registration application document in rd is dormant,
- $is\_taxed(apid, s)$ $\rightarrow$ in situation s the service charge for applicant with id $apid$ is taxed,
- $avail\_taxno(apid, s)$ $\rightarrow$ in situation s tax information is available for applicant with id $apid$,
- $charged(apid, s)$ $\rightarrow$ in situation s the applicant with $apid$ is charged,
- $registered(apid, s)$ $\rightarrow$ in situation s the applicant identified with $apid$ is successfully registered for a business,
- $located(apid, s)$ $\rightarrow$ in situation s the address of applicant with id $apid$ can be located,
- $notified(apid, s)$ $\rightarrow$ in situation s notification is sent to applicant with id $apid$.

Action Precondition Axioms

We recall that preconditions for the primitive actions are expressed in action precondition axioms which take the general form:

$$Poss(a, s) \supset necessary\_conditions$$

The essential primitive actions are axiomatized as follows[6]:

$$Poss(receive(rd), s) \supset exist\_rd(s) \wedge \neg received(rd, s) \tag{5.2.1}$$

$$Poss(return(rd), s) \supset \top(true) \tag{5.2.2}$$

$$Poss(check\_syntax(rd), s) \supset exist\_rd(s) \tag{5.2.3}$$

$$Poss(validate\_rd(rd), s) \supset exist\_rd(s) \tag{5.2.4}$$

[6]The parameter $rdms$ is a shorthand for backend record and document management system.

$$Poss(create\_rdbo(rd), s) \supset logged\_on(rdms, s) \wedge accessible(rdms, s) \wedge exist\_rd(s) \wedge \neg has\_rdbo(rd, s) \tag{5.2.5}$$

$$Poss(store\_rdbo, s) \supset logged\_on(rdms, s) \wedge accessible(rdms, s) \wedge writable(rdms, s) \tag{5.2.6}$$

$$Poss(retrieve\_rdbo(id), s) \supset has\_rdbo(rd, s) \wedge accessible(rdms, s) \tag{5.2.7}$$

$$Poss(login\_crm, s) \supset accessible(crm, s) \tag{5.2.8}$$

$$Poss(access\_crm, s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge has\_data(crm, s) \tag{5.2.9}$$

$$Poss(login\_fi, s) \supset accessible(fi, s) \tag{5.2.10}$$

$$Poss(access\_fi, s) \supset logged\_on(fi, s) \wedge accessible(fi, s) \wedge has\_data(fi, s) \tag{5.2.11}$$

$$Poss(lookup\_cusprof(apid), s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge has\_data(crm, s) \wedge has\_cusprofbo(apid, s) \tag{5.2.12}$$

$$Poss(create\_cusprofbo(apid), s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge has\_data(crm, s) \wedge \neg has\_cusprofbo(apid, s) \tag{5.2.13}$$

$$Poss(add\_cusprofbo(val), s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge has\_cusprofbo(apid, s) \tag{5.2.14}$$

$$Poss(store\_cusprofbo, s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge writable(crm, s) \tag{5.2.15}$$

$$Poss(display\_cusprof(apid), s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge has\_cusprofbo(apid, s) \tag{5.2.16}$$

$$Poss(lookup\_binfo(apid), s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge has\_cusprofbo(apid, s) \wedge logged\_on(fi, s) \wedge accessible(fi, s) \wedge has\_data(fi, s) \tag{5.2.17}$$

$$Poss(display\_binfo(apid), s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge has\_cusprofbo(apid, s) \wedge logged\_on(fi, s) \wedge accessible(fi, s) \wedge has\_binfobo(apid, s) \tag{5.2.18}$$

$$Poss(create\_binfobo(apid), s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge has\_cusprofbo(apid, s) \wedge logged\_on(fi, s) \wedge accessible(fi, s) \wedge has\_data(fi, s) \wedge \neg has\_binfobo(apid, s) \tag{5.2.19}$$

$$Poss(add\_binfobo(val), s) \supset logged\_on(crm, s) \wedge accessible(crm, s) \wedge has\_cusprofbo(apid, s) \wedge logged\_on(fi, s) \wedge accessible(fi, s) \wedge has\_binfobo(apid, s) \tag{5.2.20}$$

$$Poss(store\_binfobo, s) \supset logged\_on(fi, s) \wedge accessible(fi, s) \wedge writable(fi, s) \tag{5.2.21}$$

$$Poss(deny, s) \supset rejectable(s) \tag{5.2.22}$$

$$\begin{aligned} Poss(archive(rd), s) \supset\ & received(rd, s) \\ & \wedge has\_rdbo(rd, s) \\ & \wedge \neg is\_dormant(rd, s) \end{aligned} \tag{5.2.23}$$

$$\begin{aligned} Poss(read\_tax(apid), s) \supset\ & logged\_on(fi, s) \\ & \wedge accessible(fi, s) \\ & \wedge avail\_taxno(apid, s) \end{aligned} \tag{5.2.24}$$

$$\begin{aligned} Poss(record\_tx(taxid), s) \supset\ & logged\_on(fi, s) \\ & \wedge accessible(fi, s) \\ & \wedge avail\_taxno(apid, s) \\ & \wedge \neg is\_taxed(apid, s) \end{aligned} \tag{5.2.25}$$

$$\begin{aligned} Poss(bill(apid), s) \supset\ & logged\_on(fi, s) \\ & \wedge accessible(fi, s) \\ & \wedge has\_binfobo(apid, s) \\ & \wedge is\_taxed(apid, s) \\ & \wedge \neg charged(apid, s) \end{aligned} \tag{5.2.26}$$

$$\begin{aligned} Poss(send\_invoice(apid), s) \supset\ & logged\_on(fi, s) \\ & \wedge accessible(fi, s) \\ & \wedge has\_binfobo(apid, s) \\ & \wedge is\_taxed(apid, s) \\ & \wedge charged(apid, s) \end{aligned} \tag{5.2.27}$$

$$\begin{aligned} Poss(notify, s) \supset\ & logged\_on(fi, s) \\ & \wedge accessible(fi, s) \\ & \wedge has\_binfobo(apid, s) \\ & \wedge is\_taxed(apid, s) \\ & \wedge charged(apid, s) \\ & \wedge registered(apid, s) \\ & \wedge located(apid, s) \\ & \wedge \neg notified(apid, s) \end{aligned} \tag{5.2.28}$$

Successor State Axioms

We recall that for each fluent a successor state is specified using successor state axiom which takes the general form:

$$\begin{aligned} Poss(a,s) \supset [&fluent(\vec{x}, do(a,s)) \equiv action\_making\_it\_true \\ &\vee a\_state\_constraint\_made\_it\_true \\ &\vee it\_was\_already\_true \\ &\wedge neither\_an\_action\_nor\_a\_state\_made\_it\_false] \end{aligned}$$

where the fluent can be in the form $fluent(\vec{x}, do(a,s))$ with arguments as shown or without argument $fluent(do(a,s))$ and with only situation.

The successor state axioms for our domain are axiomatized as follows:

$$\begin{aligned} Poss(a,s) \supset [&exist\_rd(do(a,s)) \equiv \\ &a = receive(rd) \\ &\vee received(rd,s) \\ &\vee exist\_rd(s) \\ &\wedge a \neq return(rd)] \end{aligned} \tag{5.2.29}$$

$$\begin{aligned} Poss(a,s) \supset [&received(rd, do(a,s)) \equiv \\ &a = receive(rd) \\ &\vee received(rd,s) \\ &\wedge a \neq return(rd)] \end{aligned} \tag{5.2.30}$$

$$\begin{aligned} Poss(a,s) \supset [&syntax\_ok(rd, do(a,s)) \equiv \\ &a = check\_syntax(rd) \\ &\vee syntax\_ok(rd,s) \\ &\wedge a \neq return(rd)] \end{aligned} \tag{5.2.31}$$

$$\begin{aligned} Poss(a,s) \supset [&is\_valid(rd, do(a,s)) \equiv \\ &a = validate\_rd(rd) \\ &\vee is\_valid(rd,s) \\ &\wedge a \neq return(rd)] \end{aligned} \tag{5.2.32}$$

$$\begin{aligned} Poss(a,s) \supset [&has\_rdbo(rd, do(a,s)) \equiv \\ &a = create\_rdbo(rd) \\ &\vee has\_rdbo(rd,s) \\ &\wedge a \neq return(rd) \wedge a \neq deny] \end{aligned} \tag{5.2.33}$$

$$\begin{aligned} Poss(a,s) \supset [&stored(rdms, do(a,s)) \equiv \\ &a = store\_rdbo \\ &\vee stored(rdms,s) \\ &\wedge a \neq deny] \end{aligned} \tag{5.2.34}$$

$$\begin{aligned} Poss(a,s) \supset [&accessible(bk, do(a,s)) \equiv \\ &a = login\_crm \vee a = login\_fi \vee a = login\_rdms \\ &\vee accessible(bk,s) \\ &\vee logged\_on(bk,s) \\ &\wedge a \neq log\_out(bk)] \end{aligned} \tag{5.2.35}$$

$$\begin{aligned} Poss(a,s) \supset [&logged\_on(bk, do(a,s)) \equiv \\ &a = login\_crm \vee a = login\_fi \vee a = login\_rdms \\ &\vee logged\_on(bk,s) \\ &\wedge a \neq log\_out(bk)] \end{aligned} \tag{5.2.36}$$

$$\begin{aligned} Poss(a,s) \supset [&has\_cusprofbo(apid, do(a,s)) \equiv \\ &a = create\_cusprofbo(apid) \\ &\vee has\_cusprofbo(apid,s) \\ &\wedge a \neq delete\_cusprofbo(apid)] \end{aligned} \tag{5.2.37}$$

$$\begin{aligned} Poss(a,s) \supset [&has\_binfobo(apid, do(a,s)) \equiv \\ &a = create\_binfobo(apid) \\ &\vee has\_binfobo(apid,s) \\ &\wedge a \neq delete\_binfobo(apid)] \end{aligned} \tag{5.2.38}$$

$$\begin{aligned} Poss(a,s) \supset [&has\_data(bk, do(a,s)) \equiv \\ &a = create\_cusprofbo(apid) \\ &\vee a = create\_binfobo(apid) \\ &\vee has\_data(bk,s) \\ &\wedge a \neq delete\_cusprofbo(apid) \wedge a \neq delete\_binfobo(apid)] \end{aligned} \tag{5.2.39}$$

$$\begin{aligned} Poss(a,s) \supset [&writable(bk, do(a,s)) \equiv \\ &(a = login\_crm \vee a = login\_fi \vee a = login\_rdms) \wedge privileged\_write(s) \\ &\vee writable(bk,s) \\ &\wedge a \neq log\_out(bk)] \end{aligned} \tag{5.2.40}$$

$$\begin{aligned} Poss(a,s) \supset [&rejectable(do(a,s)) \equiv \\ &a = deny \\ &\vee rejectable(s) \\ &\wedge a \neq notify] \end{aligned} \tag{5.2.41}$$

$$\begin{aligned} Poss(a,s) \supset [&is\_dormant(rd, do(a,s)) \equiv \\ &a = archive(rd) \\ &\vee is\_dormant(rd,s) \\ &\wedge a \neq reactivate] \end{aligned} \tag{5.2.42}$$

$$\begin{aligned} Poss(a,s) \supset [&is\_taxed(apid, do(a,s)) \equiv \\ &a = read\_tax(apid) \wedge record\_tx(taxid) \\ &\vee is\_taxed(apid,s) \\ &\wedge a \neq deny] \end{aligned} \tag{5.2.43}$$

$$\begin{aligned} Poss(a,s) \supset [&avail\_taxno(apid, do(a,s)) \equiv \\ &a = read\_tax(apid) \\ &\vee avail\_taxno(apid,s) \vee is\_taxed(apid,s) \\ &\vee \neg is\_dormant(rd,s)] \end{aligned} \tag{5.2.44}$$

$$\begin{aligned} Poss(a,s) \supset [&charged(apid, do(a,s)) \equiv \\ &a = bill(apid) \\ &\vee charged(apid,s) \\ &\wedge a \neq deny] \end{aligned} \tag{5.2.45}$$

$$\begin{aligned} Poss(a,s) \supset [&registered(apid, do(a,s)) \equiv \\ &a = bill(apid) \wedge payment\_trans(s) \\ &\vee registered(apid,s) \\ &\wedge charged(apid,s) \\ &\wedge \neg rejectable(s)] \end{aligned} \tag{5.2.46}$$

$$\begin{aligned} Poss(a,s) \supset [&located(apid, do(a,s)) \equiv \\ &a = display\_cusprof(apid) \\ &\vee located(apid,s) \\ &\wedge \neg moved(apid,s)] \end{aligned} \tag{5.2.47}$$

$$\begin{aligned} Poss(a,s) \supset [&notified(apid, do(a,s)) \equiv \\ &a = send\_invoice(apid) \vee a = notify \\ &\vee registered(apid,s) \\ &\wedge notified(apid,s)] \end{aligned} \tag{5.2.48}$$

Effect Axioms

We recall that for a certain action, an effect axiom states that if $Poss(a,s)$, i.e., when it is possible to perform a in situation s under some *conditions*, then fluent will be true resulting from doing a in s. Effect axioms take this general form:

$$Poss(a,s) \wedge conditions \supset fluent(\vec{x}, do(a,s))$$

Some relevant effect axioms are axiomatized as follows:

$$Poss(a,s) \wedge a = deny \vee a = return(rd) \supset rejectable(do(a,s)) \tag{5.2.49}$$

$$Poss(a, s) \wedge (a = login\_crm \vee a = login\_fi) \supset accessible(bk, do(a, s)) \tag{5.2.50}$$

$$Poss(a, s) \wedge a = bill(apid) \supset charged(apid, do(a, s)) \tag{5.2.51}$$

$$Poss(a, s) \wedge (a = bill(apid) \wedge send\_invoice(apid) \wedge notify) \supset registered(apid, do(a, s)) \tag{5.2.52}$$

Axioms for Initial Situation

We also recall that for a basic action theory of a domain, we specify axioms for the initial situation S_0 which state what is known of the truth value of predicates and fluents relativized to the initial situation S_0:

$$exist\_rd(S_0) \wedge received(rd, S_0) \wedge syntax\_ok(rd, S_0) \wedge is\_valid(rd, S_0) \tag{5.2.53}$$

$$\begin{aligned} &\neg is\_dormant(rd, S_0) \wedge \neg charged(apid, S_0) \wedge \\ &\neg registered(apid, S_0) \wedge \neg notified(apid, S_0) \wedge \neg rejectable(S_0) \end{aligned} \tag{5.2.54}$$

$$\begin{aligned} &logged\_on(crm, S_0) \wedge accessible(crm, S_0) \wedge \\ &logged\_on(fi, S_0) \wedge accessible(fi, S_0) \wedge \\ &logged\_on(rdms, S_0) \wedge accessible(rdms, S_0) \end{aligned} \tag{5.2.55}$$

$$\begin{aligned} &writable(crm, S_0) \wedge writable(fi, S_0) \wedge \\ &writable(rdms, S_0) \wedge privileged\_write(S_0) \end{aligned} \tag{5.2.56}$$

5.2.3 Model-based Program

A Model-based program is an instance of GOLOG program that is based on the basic action theory domain model M that we have developed in the previous section. We recall the fundamental constructs and formal semantics of GOLOG program which are described in section 4.2.4 in chapter 4. Consequently a GOLOG program consists of a set of procedural complex actions such as those we have formalized in table 5.11 where each procedural complex action can be implemented using the building blocks of the basic action theory in $\mathcal{L}_{sitcalc}$ for our business registration scenario. In definition (4.2.39) we have defined that a composition of enterprise service is obtained from an action execution trajectory $\vec{A} = [a_1, \ldots, a_n]$ which is a model-based program instance and the trajectory represents a solution to a web service composition problem iff

$$M \models Do(\delta, S_0, do([a_1, \ldots, a_n], S_0))$$

Generation of such an instance $\vec{A}$ from the model M is the theorem proving task to prove that the basic action theory entails:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una}$$

$$\mathcal{D} \models \exists s'.Do(\delta, S_0, do(\vec{A}, S_0))$$

An agent is capable of executing world-altering and knowledge-gathering actions accordingly given an ordered generated execution plan for these actions. The complex actions for the corresponding service operations can be implemented with the situation term as shown in the axiomatizations abbreviated. A fluent is

written in this shorthand form without situation s and can be *macro-expanded* to restore the situation term if necessary [Levesque et al., 1997, Reiter, 2001a]. Procedures that implement the complex actions are shown as follows:

---

proc
RecvRegApplDoc$(rd, type, loc, ind)$
$\forall(rd)[exist\_rd \vee received\_rd]?;$
$check\_syntax(rd); validate\_rd(rd);$
$\pi(rd)[syntax\_ok(rd) \wedge is\_valid(rd) \wedge \neg has\_rdbo(rd)]?;$ (5.2.57)
$create\_rdbo(rd); store\_rdbo \,||$
$\pi(rd)[\neg has\_rdbo(rd) \vee \neg stored(rdms) \vee$
$\neg syntax\_ok(rd) \vee \neg is\_valid(rd)]?; return(rd).$
endproc

---

proc
SearchApplProfile$(apid)$
$login\_crm;$
$\forall(apid)$
while $((\exists apid).has\_cusprofbo(apid) \wedge [access\_crm]?)$ **do** (5.2.58)
$lookup\_cusprof(apid);$
$display\_cusprof(apid);$
endwhile
endproc

---

proc
CreateApplProfile$(apid)$
$login\_crm;$
if
$[access\_crm]? \,|\, \forall(apid)(\neg\exists apid \in has\_cusprofbo(apid))$ (5.2.59)
then
$create\_cusprofbo(apid);$
endif
endproc

---

proc
SearchApplBankInfo$(apid)$
$login\_fi$;
$\forall(apid)$
while $((\exists apid).has\_binfobo(apid) \wedge [access\_fi]?)$ **do** (5.2.60)
$lookup\_binfo(apid)$;
$display\_binfo(apid)$;
endwhile
endproc

---

proc
CreateApplBankInfo$(apid)$
$login\_fi$;
if
$[access\_fi]? \,|\, \forall(apid)(\neg\exists apid \in has\_binfobo(apid))$ (5.2.61)
then
$create\_binfobo(apid)$;
endif
endproc

---

proc
RejBusiReg$(apid, rd)$
if
$[rejectable]?$
then
$deny; return(rd)$; (5.2.62)
call proc $ArchiveReg(rd)$ $||$
$(login\_crm; lookup\_cusprof(apid); add\_cusprofbo(deny))$
endif
endproc

---

proc
RecApplTaxInfo$(apid, rd, binfobo)$
$login\_crm \,||\, login\_fi \,||\, read\_tax(apid)$
if
$[\neg is\_taxed \wedge avail\_taxno(apid)]?$
then
$lookup\_cusprof(apid) \,||\, lookup\_binfo(apid)$
$record\_tx(binfobo); add\_cusprofbo(taxed)$
endif
endproc

(5.2.63)

---

proc
BillAppl$(apid, binfobo)$
$login\_crm \,||\, login\_fi$
if
$[\neg charged(apid) \wedge \neg registered(apid)]?$
then
$lookup\_cusprof(apid); bill(apid); send\_invoice(apid);$
$store\_cusprofbo \,||\, store\_binfobo$
endif
endproc

(5.2.64)

---

proc
ConfirmReg$(apid)$
$login\_crm;$
if
$[registered(apid) \wedge located(apid)]?$
then
$notify; store\_cusprofbo$
endif
endproc

(5.2.65)

$$
\begin{array}{l}
\textbf{proc} \\
\quad \textbf{ArchiveReg}(rd) \\
\qquad \textbf{if} \\
\qquad\quad [\neg is\_dormant(rd) \wedge (registered(apid) \vee rejectable)]? \\
\qquad \textbf{then} \\
\qquad\quad archive(rd); \\
\qquad \textbf{endif} \\
\textbf{endproc}
\end{array}
\tag{5.2.66}
$$

$$
\begin{array}{l}
\textbf{proc} \\
\quad \textbf{main\_controller}() \\
\qquad \textbf{while}\ ((\exists\, apid).\neg registered(apid) \vee \neg rejectable)\ \textbf{do} \\
\qquad\quad [(\pi\, apid, rd)\, RecvRegApplDoc; \\
\qquad\quad (SearchApplProfile(apid) \,|\, CreateApplProfile(apid)); \\
\qquad\quad (SearchApplBankInfo(apid) \,|\, CreateApplBankInfo(apid)); \\
\qquad\quad RecApplTaxInfo(apid); BillAppl(apid); ConfirmReg(apid); ArchiveReg(rd)] \\
\qquad\quad \langle\rangle\ < rejectable \rightarrow RejBusiReg(apid, rd) > \\
\qquad \textbf{endwhile} \\
\textbf{endproc}
\end{array}
\tag{5.2.67}
$$

As a matter of fact, these GOLOG/ConGolog procedures can be directly mapped to GOLOG/ConGolog programs using the Prolog programming language syntax. For parsimonious reasons, it is chosen to show the more neat and brief high level procedures using the expression syntax which can be referenced in [Reiter, 2001a]. Readers should have no problem to map these procedures to code easily and an excerpt is shown in listing C.1 in appendix C.

5.3 Discussion

The main controller in listing (5.2.67) serves as a *controller service* for the composite service which supports rejection which is modeled as exogenous actions in the ConGolog language.

By and large one can compare this approach of high level logic programming based on the situation calculus action theory to the Roman approach described in section 4.3 in chapter 4 in the following generalized aspects:

- a high level concept of *community ontology* which represents the common understanding of a shared conceptualized reference semantics between services, regarding the meaning of the offered operations, the semantics of the data flowing through the service operations;
- a set of *available services* which are the actual operational units available in a community;
- a *mapping* from the available services to the community ontology which describes how service behavior is exposed in terms of the community ontology;
- and a client request specification which expresses the client's functional or non-functional requirements that are articulated using the community ontology.

In the approach of the Roman Model, the community ontology is the set of *actions* shared among the action alphabet; the available web services are those services in the service community; the mapping from the available services to the community ontology is represented by the finite state transition systems that represent the available services and the distinct client request specification is the target service. The Roman approach is built upon foundations that mainly takes into account the client, independent from the available services. Thus it can be regarded as a *client driven approach*.

Comparing to the Roman approach, the current applied approach in this chapter which originates from the seminal research works by Reiter [Reiter, 2001a], McIlraith et al., represents a slightly different view of the problem by doing without a distinct client specification. The community ontology is derived mainly from the account of the available services by suitably coordinating and reconciling them. Therefore it can be viewed as a *service centric approach*. Services are seen as transition systems and the common community ontology is the basic action theory in the situation calculus. Having the advantage of high expressiveness in characterizing dynamical systems, service names and concepts mapping in the common ontology is associated with a generic procedure (service) that is a GOLOG/ConGolog procedural program. The client's service request specification is in fact yet another GOLOG/ConGolog program which acts as controller and with the help of agent executable actions, it specifies acceptable sequences of actions for the agent to execute. However this does not represent a transition system that the client wants to realize nor can it influence its realization much.

For our scenario, due to the presence of a comprehensive process oriented service configuration backdrop, it is not necessary to characterize additional client request specification. In fact in the enterprise services world, the complex and often heavyweight services are better treated with the service centric approach because oftentimes exisiting available services can be combined suitably to achieve a synthesized set of functional requirements that can already suffice a range of desired goals.

6 Conclusion and Outlook

In this thesis we study integration of existing SAP enterprise services in a guiding business process scenario for SOA4All. Knowing the goals of the SOA4All project [Domingue et al., 2008] to provide a service delivery platform to integrate vast number of existing services on the web for both expert and laymen. We have studied the possibility of integrating existing SAP enterprise services of the SAP NetWeaver Suites products within the current project context. With the main idea of SOA4All to integrate such wide-band of differently conditioned services, from simple toy services to transaction-aware enterprise service, which often have very different semantics, information requirements in terms of inputs and outputs and inherent constraints, we are assured that the complex enterprise services require semantically expressive framework to handle integration sufficiently.

6.1 Summary

We have given characterization of the fundamentals of process oriented enterprise services and answered the following:

a) What is the technical nature of enterprise services with their underlying process models and constraints?

b) What information model and behavior model exist for SAP enterprise processes that are relevant to our business scenario?

c) How are process constraints characterized and enforced when process activities perform actions that change process states?

d) How can one compose desired functionality from existing enterprise services to satisfy process goal with due observance to process constraints?

e) What sufficiently expressive formalism and semantic framework can be used to characterized dynamical and automatic composition of enterprise services?

In a guiding business process integration scenario we have identified that the desired process can be realized functionally by a set of existing enterprise services, therefore we realize that in fact our *integration scenario* of existing enterprise services can be *reduced* to the well-known and sufficiently researched *service composition problem as a means of functional synthesis*.

We have given brief descriptions and in-depth literature summary to relevant existing approaches and practices from industry and academia researching the issues of service modeling, matching, discovery and composition. We deliver a critical appraisal of these approaches based on selected dimensions and relevant properties on completeness, correctness, observability, type of interaction model, client involvement and handling of process artifacts such as dataflows, control flows.

Knowing of the appropriateness of formal methods of knowledge representation such as the situation calculus [Reiter, 2001a] for reasoning in terms of expressiveness, completeness, decidability and favorable runtime properties, we further internalize works in the domain of intelligent rational agent programming

theory for composition of services. We are convinced that we can approach the current integration problem at best with this means.

We have laid down grounded logical foundations for

I) formalized service selection based on signature with observance to covariant and contravariant type handling issues;

II) formal approach of semantic exposition of service capabilities, functional and non-functional properties;

III) studying semantic matching of services based on semantical capability descriptions;

IV) fundamentals of the situation calculus as a useful action theory to model dynamical system and the related logic programming languages GOLOG/ConGolog;

V) an in-depth survey of an automatic service composition framework that is able to admit client functional specification and capable of automatic service synthesis based on proper and correct delegation of actions to available services.

Altogether they represent useful tools at our disposal to tackle our problem.

We have shown an appropriate axiomatization of the guiding business process scenario using the situation calculus basic action theory, thereby we have modeled the process as a domain and enterprise service operations as world-altering operations in the situation calculus action theory. We have axiomatized the inherent business process constraints with

i) situation calculus axioms of the initial states;

ii) action precondition axioms for each executable actions (operations);

iii) effects axioms;

iv) successor state axioms;

v) domain independent axioms.

With the approach initially proposed Reiter and McIlraith for service functional synthesis through adapting and composing GOLOG/ConGolog generic procedures to mimic an agent based service controller to realize the composition, we have provided the GOLOG/ConGolog realization of our domain action theory given the fact that agents are capable of executing knowledge-gathering actions to realize our composition plan derived from the resulting sequence of executable actions which represents the order in which the service operations are executed in our domain.

6.2 Discussion of Lesson Learned

First of all the current enterprise services are not semantically annotated. In fact semantic enterprise services are not prevailing in the enterprise service arena due to many reasons. We do not assume that existing integration approaches are inferior, nevertheless if there were the necessary semantic information in form of metadata to explicate process intrinsic semantics, if process constraints were formulated explicitly, we would have had a shortcut to *real integration* in the enterprise in two ways: (i) services would be more available due to better service description, discovery and selection utilizing proven logical reasoning and inference

mechanisms to locate services. (ii) Services would be more interoperable due to possibility to use intelligent service composition and synthesis techniques as we have reviewed to automatically combine existing services to achieve more complex goals. The ubiquitous challenge of the project lies in the fact that we constantly lack information which we require in the course of an attempt to axiomatize our process in a domain theory. Existing business process integration approach is heavily based on syntactic language such as BPEL and WS-BPEL which lack well-defined semantics regarding intrinsic process constaints and therefore cannot suffice the requirement of automatic process composition without intervention. Since the industry has not moved forward in a direction that favors the more expressive and knowledgeable approach which we desire.

No business process management software vendor has seriously offered more intelligent process products yet. Consequently, process integration often stays in a low level in terms of automatism and intelligence. We still have to append semantic metadata to existing services so that we can characterize them accordingly. It is obvious that such manual approach is to a large extent tedious and it is impractical to annotate every SAP enterprise service this way because of the sheer number of available services; nor is it necessary to do so. If a more favorable development in future release would incorporate semantics explication at design time, it would definitely enhance the value and usability of the SAP enterprise applications as a whole.

Nevertheless serious domain knowledge is required in the substance and topics which we have discussed extensively in this thesis. In order to strike a balanced solution to current process integration, our conceptual approach serves as a proof-of-concept demonstration that a formalized and yet very expressive approach can handle many of the heavyweight, implicit semantics of the enterprise services better. At the core it helps to achieve our goal. Nevertheless we also realize that we must strike a balance between expressiveness and the computational properties of such an approach, so that we can still ensure decidability.

One of the advantages of our conceptual approach in real integration is that services as generic procedures can be easily reused. We can store them together with the existing services and access them whenever necessary. From our formal characterization of service matching, we can obtain more precise matches to service queries. We believe that that business process experts should understand these complex issues when they approach a task to reengineer processes or combine services before making the conclusion that one certain widely used process language alone will be a panacea to all integration problems.

6.3 Outlook

There are several ways to improve the current approach presented here. Regarding the current composition approach to integrate existing services, future research can be stimulated to

- *incorporate dataflow*: a dataflow is a network of concurrently executing processes or automata that communicate by exchanging messages over message passing channels. It represents the routes which data can take during process execution. In order to take dataflow into account, the current service composition framework based on action theory of the situation calculus and GOLOG logic program can be extended in a way that can handle dataflow in processes. For instance one possible approach to incorporate handling of dataflow is to model dataflow as concurrent, first-class actions which can be executed interleaved with world-altering actions. This approach resembles the interleaved concurrency of knowledge-gathering actions executed by an agent [McIlraith & Son, 2002]; likewise an agent can execute concurrently dataflow actions representing concurrent processes with message passing. A dedicated channel connecting processes which serves as a route for a certain dataflow can be modeled using a *relational fluent* with an appropriate arity: the first argument, for instance, can represent a source process that is also modeled as a primitive action, the second argument takes the target (pro-

cess), the dataflow object can be referenced in the subsequent argument with the last argument having type situation by conventional. When a dataflow process engages an channel, i.e., passes through the channel, it induces the relational fluent representing the channel to become true. Other dataflow sharing the same channel will not interfere with the existing occupant since a different dataflow object is passed to the relational fluent. Whenever the channel is idle or dataflow has stop using the channel, the fluent has a false value;

- *automatic generation of process execution language artifacts*: from a resulting execution plan obtained from the agent executing the actions over a situation tree, a mapping should be defined that allow to output the actual process or service to a execution plan that is represented by certain templates. A generator for instance can read a set of sequences of these actions from GOLOG/ConGolog interpreter and fill in the templates in order to output the execution plan in a concrete syntax, for instance, generating WS-BPEL artifacts so that a BPEL engine can use it as immediate input to execute the sequence of services. Since GOLOG/ConGolog is high level program that captures procedural knowledge of how to accomplish a composition task through the entailed sequence of generic procedures representing service actions without having to specify all system specific details; such programs are too abstract for execution directly and must therefore appeal to agent instantiating and executing them with a known state of the world. Consequently, automatic generating these implicitly captured procedural knowledge into business process execution language output can instantaneously enable an execution engine of that language to execute the related services, as a matter of fact, it is a reasonable step to bridge the knowledge representation formalism with more widely adopted tools;

- *support client request specification*: in the current approach client specification is not distinctly accounted for. In contrary to the Roman approach where client specification of the desired target service represents a central concept in service composition. The current approach does not distinguish this, instead the client service specification is exactly the GOLOG/ConGolog program which specifies acceptable sequence of actions as in AI planning. However it is not necessarily the situation based transition system which a client intends to realize. The advantage of having dedicated client request in form of a specification can enable a composition transition system to be synthesized exactly as desired by the client as described in the Roman approach. Another advantage is the possibility to allow client specification to be incompletely specified (as a form of desired non-determinism by the client) so that a composition system can take more responsibility to synthesize a less strictly specified request. Such a property in client specification proves to be helpful in dealing with non-deterministic situations regarding uncertainty in outcomes of certain operation executions of services;

- *better handle failure in behavior composition*: failure situations are inherent to real world services. The fact that failure can strike asynchronously has a significant impact on modeling behavior of composition. Exogenous actions are suitable in modeling failure in terms of generalizing failures as asynchronously arriving events. The peril of this model is that it will require interrupts to handle the failure, which in ConGolog is a (possibly prioritized) loop that repeatedly executes a partial program once an interrupt gets control until a test condition becomes false. The caveat thereby is that first when the interrupt finishes and releases control, any actions that are executable can execute. This property impacts failure handling in the sense that control is not resumed after handling is finished where the procedure left off but instead any possible concurrently executed actions. Secondly, to guarantee terminating the loop corresponding to an interrupt we need extra interrupt enabling and disabling actions which have the effects of block all other interrupts with lower priority. Consequence of this property can impact the runtime efficiency of failure handling with interrupts.

These points mentioned represent possible directions for extending the current approach of situation calculus theory on composition. It seems that the first and second point are of utmost interest to business processes

while the last two can be complemented with features by other frameworks.

A GOLOG Interpreter

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%                   A GOLOG INTERPRETER IN SWI-PROLOG
%
% This software was developed by the Cognitive Robotics Group under the
% direction of Hector Levesque and Ray Reiter.
%
%        Do not distribute without permission.
%        Include this notice in any copy made.
%
%
%         Copyright (c) 1992-1997 by The University of Toronto,
%                        Toronto, Ontario, Canada.
%
%                          All Rights Reserved
%
% Permission to use, copy, and modify, this software and its
% documentation for research purpose is hereby granted without fee,
% provided that the above copyright notice appears in all copies and
% that both the copyright notice and this permission notice appear in
% supporting documentation, and that the name of The University of
% Toronto not be used in advertising or publicity pertaining to
% distribution of the software without specific, written prior
% permission.  The University of Toronto makes no representations about
% the suitability of this software for any purpose.  It is provided "as
% is" without express or implied warranty.
%
% THE UNIVERSITY OF TORONTO DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
% SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
% FITNESS, IN NO EVENT SHALL THE UNIVERSITY OF TORONTO BE LIABLE FOR ANY
% SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER
% RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
% CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
% CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.*/
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- style_check(-discontiguous).
:- set_prolog_flag(optimise, true).


:- dynamic proc/2,
      restoreSitArg/3.          /* Compiler directives. Be sure   */

:- op(800, xfy, [&]).   /* Conjunction */
:- op(850, xfy, [v]).   /* Disjunction */
:- op(870, xfy, [=>]).  /* Implication */
:- op(880,xfy, [<=>]).  /* Equivalence */
:- op(950, xfy, [:]).   /* Action sequence */
:- op(960, xfy, [#]).   /* Nondeterministic action choice */

do(E1 : E2,S,S1) :- do(E1,S,S2), do(E2,S2,S1).
do(?(P),S,S) :- holds(P,S).
do(E1 # E2,S,S1) :- do(E1,S,S1) ; do(E2,S,S1).
```

```
do(if(P,E1,E2),S,S1) :- do((?(P) : E1) # (?(-P) : E2),S,S1).
do(star(E),S,S1) :- S1 = S ; do(E : star(E),S,S1).
do(while(P,E),S,S1):- do(star(?(P) : E) : ?(-P),S,S1).
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).
do(E,S,S1) :- proc(E,E1), do(E1,S,S1).
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).

/* introduced a predicate to handle temporal description by Reiter */
start(do(A,S),T) :- time(A,T).

/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New. */

sub(_,_,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- \+ var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- \+ T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                    T2 =..[F|L2].
sub_list(_,_,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

/* The holds predicate implements the revised Lloyd-Topor
   transformations on test conditions.  */

holds(P & Q,S) :- holds(P,S), holds(Q,S).
holds(P v Q,S) :- holds(P,S); holds(Q,S).
holds(P => Q,S) :- holds(-P v Q,S).
holds(P <=> Q,S) :- holds((P => Q) & (Q => P),S).
holds(-(-P),S) :- holds(P,S).
holds(-(P & Q),S) :- holds(-P v -Q,S).
holds(-(P v Q),S) :- holds(-P & -Q,S).
holds(-(P => Q),S) :- holds(-(-P v Q),S).
holds(-(P <=> Q),S) :- holds(-((P => Q) & (Q => P)),S).
holds(-all(V,P),S) :- holds(some(V,-P),S).
holds(-some(V,P),S) :- \+ holds(some(V,P),S).  /* Negation */
holds(-P,S) :- isAtom(P), \+ holds(P,S).       /* by failure */
holds(all(V,P),S) :- holds(-some(V,-P),S).
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).

/* The following clause treats the holds predicate for non fluents, including
   Prolog system predicates. For this to work properly, the GOLOG programmer
   must provide, for all fluents, a clause giving the result of restoring
   situation arguments to situation-suppressed terms, for example:
         restoreSitArg(ontable(X),S,ontable(X,S)).            */

holds(A,S) :- restoreSitArg(A,S,F), F ;
              \+ restoreSitArg(A,S,F), isAtom(A), A.

isAtom(A) :- \+ (A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
     A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W)).

%restoreSitArg(poss(A),S,poss(A,S)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EOF: golog.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing A.1: A GOLOG interpreter implemented in SWI Prolog

B ConGolog Interpreter

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               A ConGolog INTERPRETER BASED ON TRANSITION SEMANTIC
%
% This software was developed by Hector J. Levesque, Yves Lesperance &
%      Giuseppe de Giacomo, Sheila McIlraith - All Rights Reserved
%
%                  It is published in the journal article:
% ConGolog: a Concurrent Programming Language Based on the Situation Calculus
%
% THE UNIVERSITY OF TORONTO DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
% SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
% FITNESS, IN NO EVENT SHALL THE UNIVERSITY OF TORONTO BE LIABLE FOR ANY
% SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER
% RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
% CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
% CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.*/
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- style_check(-discontiguous).
:- set_prolog_flag(optimise, true).
:- dynamic proc/2,
     restoreSitArg/3.          /* Compiler directives. Be sure  */
:- op(800, xfy, [&]).   /* Conjunction */
:- op(850, xfy, [v]).   /* Disjunction */
:- op(870, xfy, [=>]).  /* Implication */
:- op(880,xfy, [<=>]).  /* Equivalence */
:- op(950, xfy, [:]).   /* Action sequence */
:- op(960, xfy, [#]).   /* Nondeterministic action choice */
/*******************************************************************/
/* Transbased ConGolog Interpreter */
/*******************************************************************/
/* trans(Prog,Sit,Prog_r,Sit_r) */
trans(act(A),S,nil,do(AS,S)) :  sub(now,S,A,AS), poss(AS,S).
trans(test(C),S,nil,S) :  holds(C,S).
trans(seq(P1,P2),S,P2r,Sr) :  final(P1,S),trans(P2,S,P2r,Sr).
trans(seq(P1,P2),S,seq(P1r,P2),Sr) :  trans(P1,S,P1r,Sr).
trans(choice(P1,P2),S,Pr,Sr) :  trans(P1,S,Pr,Sr) ; trans(P2,S,Pr,Sr).
trans(pick(V,P),S,Pr,Sr) :  sub(V,_,P,PP), trans(PP,S,Pr,Sr).
trans(iter(P),S,seq(PP,iter(P)),Sr) :  trans(P,S,PP,Sr).
trans(if(C,P1,P2),S,Pr,Sr) :  holds(C,S),trans(P1,S,Pr,Sr) ; holds(neg(C),S),trans(P2,S,Pr,Sr).
trans(while(C,P),S,seq(PP,while(C,P)),Sr) :  holds(C,S),trans(P,S,PP,Sr).
trans(conc(P1,P2),S,conc(P1r,P2),Sr) :  trans(P1,S,P1r,Sr).
trans(conc(P1,P2),S,conc(P1,P2r),Sr) :  trans(P2,S,P2r,Sr).
trans(prconc(P1,P2),S,prconc(P1r,P2),Sr) :  trans(P1,S,P1r,Sr).
trans(prconc(P1,P2),S,prconc(P1,P2r),Sr) :  not trans(P1,S,_,_),trans(P2,S,P2r,Sr).
trans(iterconc(P),S,conc(PP,iterconc(P)),Sr) :  trans(P,S,PP,Sr).
trans(pcall(P_Args),S,Pr,Sr) :  sub(now,S,P_Args,P_ArgsS), proc(P_ArgsS,P), trans(P,S,Pr,Sr).

/* final(Prog,Sit) */
final(nil,S).
final(seq(P1,P2),S) :  final(P1,S),final(P2,S).
```

```
final(choice(P1,P2),S) :  final(P1,S) ; final(P2,S).
final(pick(V,P),S) :  sub(V,_,P,PP), final(PP,S).
final(iter(P),S).
final(if(C,P1,P2),S) :  holds(C,S),final(P1,S) ; holds(neg(C),S),final(P2,S).
final(while(C,P),S) :  holds(neg(C),S) ; final(P,S).
final(conc(P1,P2),S) :  final(P1,S),final(P2,S).
final(prconc(P1,P2),S) :  final(P1,S),final(P2,S).
final(iterconc(P),S).
final(pcall(P_Args)) :  sub(now,S,P_Args,P_ArgsS), proc(P_ArgsS,P),final(P,S).

/* trans*(Prog,Sit,Prog_r,Sit_r) */
trans*(P,S,P,S).
trans*(P,S,Pr,Sr) :  trans(P,S,PP,SS), trans*(PP,SS,Pr,Sr).

/* do(Prog,Sit,Sit_r) */
do(P,S,Sr) :  trans*(P,S,Pr,Sr),final(Pr,Sr).

/* holds(Cond,Sit): as defined in [34] */
holds(and(F1,F2),S) :  holds(F1,S), holds(F2,S).
holds(or(F1,F2),S) :  holds(F1,S); holds(F2,S).
holds(all(V,F),S) :  holds(neg(some(V,neg(F))),S).
holds(some(V,F),S) :  sub(V,_,F,Fr), holds(Fr,S).
holds(neg(neg(F)),S) :  holds(F,S).
holds(neg(and(F1,F2)),S) :  holds(or(neg(F1),neg(F2)),S).
holds(neg(or(F1,F2)),S) :  holds(and(neg(F1),neg(F2)),S).
holds(neg(all(V,F)),S) :  holds(some(V,neg(F)),S).
holds(neg(some(V,F)),S) :  not holds(some(V,F),S). /* Negation by failure */

holds(P_Xs,S) :  P_Xs\=and(_,_),P_Xs\=or(_,_),P_Xs\=neg(_),P_Xs\=all(_,_),P_Xs\=some(_.
    _), sub(now,S,P_Xs,P_XsS), P_XsS.
holds(neg(P_Xs),S) :  P_Xs\=and(_,_),P_Xs\=or(_,_),P_Xs\=neg(_),P_Xs\=all(_,_),P_Xs\=
    some(_._), sub(now,S,P_Xs,P_XsS), not P_XsS. /* Negation by failure */

/* sub(Const,Var,Term1,Term2): as defined in [34] */
sub(X,Y,T,Tr) :  var(T), Tr = T.
sub(X,Y,T,Tr) :  not var(T), T = X, Tr = Y.
sub(X,Y,T,Tr) :  T \= X, T =..[F|Ts], sub_list(X,Y,Ts,Trs), Tr =..[F|Trs].
sub_list(X,Y,[],[]).
sub_list(X,Y,[T|Ts],[Tr|Trs]) :  sub(X,Y,T,Tr), sub_list(X,Y,Ts,Trs).

/* McIlraith and Son's extensions */
/* User Constraints Customization */
trans(A,S,R,S1) :- primitive_action(A), (Poss(A,S), desirable(A,S), R=nil, S1=do(A,s));
    fail.
desirable(A,S) :- \+ not_desirable(A,S).

/* order connective construct */
final(P:A,S)      :- action(A), final([P,achieve(poss(A),0),A],S).
trans(P:A,S,R,S1) :- action(A), trans([P,achieve(poss(A),0),A],S,R,S1).

/* sensing actions */
holds(f(X), do(a(X), S)) :- exec(a(X),S).
exec(a(X),S) :- <external call>
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EOF: congolog.pl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing B.1: A ConGolog interpreter implemented published in [Giacomo et al., 2000]

C ConGolog Model-Based Program Instance

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%                A ConGolog Model-Based Program Instance
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% primitive action declarations
%
primitive_action(receive(Rd)).
primitive_action(return(Rd)).
primitive_action(check_syntax(Rd)).
primitive_action(validate_rd(Rd)).
primitive_action(create_rdbo(Rd)).
primitive_action(store_rdbo).
primitive_action(retrieve_rdbo(Id)).
primitive_action(login_crm).
primitive_action(access_crm).
primitive_action(login_fi).
primitive_action(access_fi).
primitive_action(lookup_cusprof(Apid)).
primitive_action(create_cusprofbo(Apid)).
primitive_action(delete_cusprofbo(Apid)).
primitive_action(add_cusprofbo(Val)).
primitive_action(store_cusprofbo).
primitive_action(display_cusprof(Apid)).
primitive_action(lookup_binfo(Apid)).
primitive_action(display_binfo(Apid)).
primitive_action(create_binfobo(Apid)).
primitive_action(delete_binfobo(Apid)).
primitive_action(add_binfobo(Val)).
primitive_action(store_binfobo).
primitive_action(deny).
primitive_action(archive(Rd)).
primitive_action(reactivate).
primitive_action(read_tax(Apid)).
primitive_action(record_tx(Taxid)).
primitive_action(bill(Apid)).
primitive_action(send_invoice(Apid)).
primitive_action(notify).
primitive_action(log_out(Bk)).

%
% fluent declarations
%
restoreSitArg(exist_rd,S,exist_rd(S)).
restoreSitArg(received(Rd),S,received(Rd,S)).
restoreSitArg(syntax_ok(Rd),S,syntax_ok(Rd,S)).
restoreSitArg(is_valid(Rd),S,is_valid(Rd,S)).
restoreSitArg(has_rdbo(Rd),S,has_rdbo(Rd,S)).
restoreSitArg(stored(Bk),S,stored(Bk,S)).
restoreSitArg(accessible(Bk),S,accessible(Bk,S)).
restoreSitArg(logged_on(Bk),S,logged_on(Bk,S)).
```

```
53 restoreSitArg(has_cusprofbo(Apid),S,has_cusprofbo(Apid,S)).
54 restoreSitArg(has_binfobo(Apid),S,has_binfobo(Apid,S)).
55 restoreSitArg(has_data(Bk),S,has_data(Bk,S)).
56 restoreSitArg(writable(Bk),S,writable(Bk,S)).
57 restoreSitArg(rejectable,S,rejectable(S)).
58 restoreSitArg(is_dormant(Rd),S,is_dormant(Rd,S)).
59 restoreSitArg(is_taxed(Apid),S,is_taxed(Apid,S)).
60 restoreSitArg(avail_taxno(Apid),S,avail_taxno(Apid,S)).
61 restoreSitArg(charged(Apid),S,charged(Apid,S)).
62 restoreSitArg(registered(Apid),S,registered(Apid,S)).
63 restoreSitArg(located(Apid),S,located(Apid,S)).
64 restoreSitArg(notified(Apid),S,notified(Apid,S)).
65 restoreSitArg(privileged_write,S,privileged_write(S)).
66
67 %
68 % initial situation declarations
69 %
70 exist_rd(s0),received(Rd,s0),syntax_ok(Rd,s0),is_valid(Rd,s0).
71 not is_dormant(Rd,s0), not charged(Apid,s0), not registered(Apid,s0),
72 not notified(Apid,s0), not rejectable(s0).
73 logged_on(Crm,s0),accessible(Crm,s0).
74 logged_on(Fi,s0),accessible(Fi,s0).
75 logged_on(Rdms,s0),accessible(Rdms,s0).
76 writable(Crm,s0),writable(Fi,s0).
77 writable(Rdms,s0),privileged_write(s0).
78
79 %
80 % action precondition axioms
81 %
82 poss(receive(Rd),S) :- exist_rd(S),not received(Rd,S).
83 poss(return(Rd),S).
84 poss(check_syntax(Rd),S) :- exist_rd(S).
85 poss(validate_rd(Rd),S) :- exist_rd(S).
86 poss(create_rdbo(Rd),S) :- logged_on(Rdms,S),accessible(Rdms,S),exist_rd(S),not
       has_rdbo(Rd,S).
87 poss(store_rdbo,S) :- logged_on(Rdms,S),accessible(Rdms,S),writable(Rdms,S).
88 poss(retrieve_rdbo(Id),S) :- has_rdbo(Rd,S),accessible(Rdms,S).
89 poss(login_crm,S) :- accessible(Crm,S).
90 poss(access_crm,S) :- logged_on(Crm,S),accessible(Crm,S),has_data(Crm,S).
91 poss(login_fi,S) :- accessible(Fi,S).
92 poss(access_fi,S) :- logged_on(Fi,S),accessible(Fi,S),has_data(Fi,S).
93 poss(lookup_cusprof(Apid),S) :- logged_on(Crm,S),accessible(Crm,S),has_data(Crm,S),
       has_cusprofbo(Apid,S).
94 poss(create_cusprofbo(Apid),S) :- logged_on(Crm,S),accessible(Crm,S),has_data(Crm,S),
       not has_cusprofbo(Apid,S).
95 poss(delete_cusprofbo(Apid),S) :- logged_on(Crm,S),accessible(Crm,S),has_data(Crm,S),
       has_cusprofbo(Apid,S).
96 poss(add_cusprofbo(Val),S) :- logged_on(Crm,S),accessible(Crm,S),has_cusprofbo(Apid,S).
97 poss(store_cusprofbo,S) :- logged_on(Crm,S),accessible(Crm,S),writable(Crm,S).
98 poss(display_cusprof(Apid),S) :- logged_on(Crm,S),accessible(Crm,S),has_cusprofbo(Apid,
       S).
99 poss(lookup_binfo(Apid),S) :- logged_on(Crm,S),accessible(Crm,S),has_cusprofbo(Apid,S),
       logged_on(Fi,S),accessible(Fi,S),has_data(Fi,S).
100 poss(display_binfo(Apid),S) :- logged_on(Crm,S),accessible(Crm,S),has_cusprofbo(Apid,S)
       ,logged_on(Fi,S),accessible(Fi,S),has_binfobo(Apid,S).
101 poss(create_binfobo(Apid),S) :- logged_on(Crm,S),accessible(Crm,S),has_cusprofbo(Apid,S
       ),logged_on(Fi,S),accessible(Fi,S),has_data(Fi,S),not has_binfo(Apid,S).
102 poss(delete_binfobo(Apid),S) :- logged_on(Crm,S),accessible(Crm,S),has_cusprofbo(Apid,S
       ),logged_on(Fi,S),accessible(Fi,S),has_data(Fi,S),has_binfo(Apid,S).
```

```
103 poss(add_binfobo(Val),S) :- logged_on(Crm,S),accessible(Crm,S),has_cusprofbo(Apid,S),
        logged_on(Fi,S),accessible(Fi,S),has_binfobo(Apid,S)
104 poss(store_binfobo,S) :- logged_on(Fi,S),accessible(Fi,S),writable(Fi,S).
105 poss(deny,S) :- rejectable(S).
106 poss(archive(Rd),S) :- received(Rd,S),has_rdbo(Rd,S),not is_dormant(Rd,S).
107 poss(reactivate,S) :- is_dormant(Rd,S).
108 poss(read_tax(Apid),S) :- logged_on(Fi,S),accessible(Fi,S),avail_taxno(Apid,S).
109 poss(record_tx((Taxid),S) :- logged_on(Fi,S),accessible(Fi,S),has_binfobo(Apid,S),
        is_taxed(Apid,S),not charged(Apid,S).
110 poss(bill(Apid),S) :- logged_on(Fi,S),accessible(Fi,S),has_binfobo(Apid,S),is_taxed(
        Apid,S),charged(Apid,S).
111 poss(send_invoice(Apid),S) :- logged_on(Fi,S).accessible(Fi,S),has_binfobo(Apid,S),
        is_taxed(Apid,S),charged(Apid,S).
112 poss(notify,S) :- logged_on(Fi,S),accessible(Fi,S),has_binfobo(Apid,S),is_taxed(Apid,S)
        ,charged(Apid,S),registered(Apid,S),located(Apid,S),not notified(Apid,S).
113 poss(log_out(Bk),S) :- logged_on(Bk,S).
114
115 %
116 % successor state axioms
117 %
118 exist_rd(do(A,S)) :- A = receive(Rd) ; received(Rd,S) ; (exist_rd(S), A \= return(Rd)).
119 received(Rd,do(A,S)) :- A = receive(Rd) ; (received(Rd,S), A \= return(Rd)).
120 syntax_ok(Rd,do(A,S)) :- A = check_syntax(Rd) ; (syntax_ok(Rd,S), A \= return(Rd)).
121 is_valid(Rd,do(A,S)) :- A = validate_rd(Rd) ; (is_valid(Rd,S), A \= return(Rd)).
122 has_rdbo(Rd,do(A,S)) :- A = create_rdbo(Rd) ; (has_rdbo(Rd,S), A \= return(Rd) , A \=
        deny).
123 stored(Rdms,do(A,S)) :- A = store_rdbo ; (stored(Rdms,S), A \= deny).
124 accessible(Bk,do(A,S)) :- A = login_crm ; A = login_fi ; A = login_rdms ; accessible(Bk
        ,S) ; (logged_on(Bk,S), A \= log_out(Bk)).
125 logged_on(Bk,do(A,S)) :- A = login_crm ; A = login_fi ; A = login_rdms ; (logged_on(Bk,
        S), A \= log_out(Bk)).
126 has_cusprofbo(Apid,do(A,S)) :- A = create_cusprofbo(Apid) ; (has_cusprofbo(Apid,S), A
        \= delete_cusprofbo(Apid)).
127 has_binfobo(Apid,do(A,S)) :- A = create_binfobo(Apid) ; (has_binfobo(Apid,S), A \=
        delete_binfobo(Apid)).
128 has_data(Bk,do(A,S)) :- A = create_cusprofbo(Apid) ; A = create_binfobo(Apid) ; (
        has_data(Bk,S), A \= delete_cusprofbo(Apid), A \= delete_binfobo(Apid)).
129 writable(Bk,do(A,S)) :- (A = login_crm ; A = login_fi ; A = login_rdms),
        privileged_write(S) ; (writable(Bk,S), A \= log_out(Bk)).
130 rejectable(do(A,S)) :- A = deny ; (rejectable(S), A \= notify).
131 is_dormant(Rd,do(A,S)) :- A = archive(Rd) ; (is_dormant(Rd,S), A \= reactivate).
132 is_taxed(Apid,do(A,S)) :- A = read_tax(Apid),record_tx(Taxid) ; (is_taxed(Apid,S), A \=
         deny).
133 avail_taxno(Apid,do(A,S)) :- A = read_tax(Apid) ; (avail_taxno(Apid,S) ; is_taxed(Apid,
        S) ; not is_dormant(Rd,S)).
134 charged(Apid,do(A,S)) :- A = bill(Apid) ; (charged(Apid,S), A \= deny).
135 registered(Apid,do(A,S)) :- A = bill(Apid),payment_trans(S) ; (registered(Apid,S),
        charged(Apid,S), not rejectable(S)).
136 located(Apid,do(A,S)) :- A = display_cusprof(Apid) ; (located(Apid,S), not moved(Apid,S
        )).
137 notified(Apid,do(A,S)) :- A = send_invoice(Apid) ; A = notify ; (registered(Apid,S),
        notified(Apid,S)).
138
139 %
140 % generalized effects
141 %
142 poss(A,S), A = deny ; A = return(Rd) :- rejectable(do(A,S)).
143 poss(A,S), (A = login_crm ; A = login_fi) :- accessible(Bk,do(A,S)).
144 poss(A,S), A = bill(Apid) :- charged(Apid,do(A,S)).
```

```
145 poss(A,S), (A = bill(Apid), A = send_invoice(Apid), A = notify) :- registered(Apid,do(A
        ,S)).
146
147 %
148 % procedure declarations
149 %
150 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
151 % RecvRegApplDoc
152 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
153 proc(recvRegApplDoc(Rd,Type,Loc,Ind), (?(all(Rd,exist_rd # received_rd))) :
        check_syntax(Rd) : validate_rd(Rd),
154 conc(
155       (
156        pi(Rd, ?(syntax_ok(Rd) & is_valid(Rd) & -has_rdbo(Rd))) :
157        pi(Rd, ?(received(Rd)) : create_rdbo(Rd)) :
158        pi(Rd, ?(received(Rd)) : store_rdbo)
159       ),
160       (
161        pi(Rd, ?(-has_rdbo(Rd) # -stored(Rdms) # -syntax_ok(Rd) # -is_valid(Rd))) :
162        pi(Rd, ?(received(Rd)) : return(Rd)
163       )
164 )).
165
166 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
167 % SearchApplProfile
168 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
169 proc(searchApplProfile(Apid),
170 login_crm,
171 (?(all(Apid))) :
172 while( ?(some(Apid, has_cusprofbo(Apid))) & test(access_crm) ) :
173         lookup_cusprof(Apid) :
174         display_cusprof(Apid)
175 ).
176
177 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
178 % CreateApplProfile
179 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
180 proc(createApplProfile(Apid),
181 login_crm,
182 if(
183     test(access_crm), (?(all(Apid, ?(some(Apid, -has_cusprofbo(Apid)))))),
184     create_cusprofbo(Apid)
185   )
186 ).
187
188 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
189 % SearchApplBankInfo
190 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
191 proc(searchApplBankInfo(Apid),
192 login_fi,
193 (?(all(Apid))) :
194 while( ?(some(Apid, has_binfobo(Apid))) & test(access_fi) ) :
195         lookup_binfo(Apid) :
196         display_binfo(Apid)
197 ).
198
199 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
200 % CreateApplBankInfo
201 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
202 proc(createApplBankInfo(Apid),
```

```
login_fi ,
if (
    test(access_fi), (?(all(Apid, ?(some(Apid, -has_binfobo(Apid)))))),
    create_binfobo(Apid)
  )
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RejBusiReg
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc(rejBusiReg(Apid,Rd),
if (
    test(rejectable),
    (
     conc(
           (deny, return(Rd), pcall(archiveReg(Rd))),
           (login_crm, lookup_cusprof(Apid), add_cusprofbo(deny))
          )
    )
  )
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RecApplTaxInfo
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc(recApplTaxInfo(Apid,Rd,Binfobo),
conc((login_crm, login_fi),read_tax(Apid)),
if (
    test(pi(Apid, (-is_taxed & avail_taxno(Apid)))),
    (
     conc(
           lookup_cusprof(Apid),
           lookup_binfo(Apid)
          ),
     record_tx(binfobo),
     add_cusprofbo(taxed)
    )
  )
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% BillAppl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc(billAppl(Apid,Binfobo),
conc(
      login_crm ,
      login_fr
     ),
if (
    test(pi(Apid, (-charged(Apid) & -registered(Apid)))),
    (
     lookup_cusprof(Apid),
     bill(Apid),
     send_invoice(Apid),
     conc(
           store_cusprofbo ,
           store_binfobo
          )
    )
  )
```

```
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ConfirmReg
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc(confirmReg(Apid),
login_crm,
if(
    test(pi(Apid, (registered(Apid) & located(Apid)))),
    (
     notify,
     store_cusprofbo
    )
   )
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ArchiveReg
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc(archiveReg(Rd),
if(
    test(pi(Rd, (-is_dormant(Rd) & (registered(Apid) # rejectable)))),
    (archive(Rd))
   )
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% controller
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc(main_controller,
while( ?(some(Apid, -registered(Apid) # -rejectable)) ) :
        prconc(
               (
                pi(Apid, pi(Rd, pcall(recvRegApplDoc(Rd,_,_,_)))),
                pi(Apid, pi(Rd, pcall(choice(
                                              pcall(searchApplProfile(Apid)),
                                              pcall(createApplProfile(Apid))
                                             )))),
                pi(Apid, pi(Rd, pcall(choice(
                                              pcall(SearchApplBankInfo(Apid)),
                                              pcall(CreateApplBankInfo(Apid))
                                             )))),
                pi(Apid, pi(Rd, pcall(recApplTaxInfo(Apid)),
                                pcall(BillAppl(Apid)),
                                pcall(confirmReg(Apid)),
                                pcall(archiveReg(Rd))
                  ))

               ),
               % exogenous interrupt
               ( if(test(rejectable), rejBusiReg(Apid,Rd)) )
              )
)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EOF: business_reg_basic_action_theory_domain_model.pl (business registration)
% Patrick Un
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing C.1: A ConGolog model-based program instance for the synthesized composite service for business registration

Bibliography

[Agarwal, 2007] Agarwal, S. (2007). *Formal Description of Web Services for Expressive Matchmaking*. PhD thesis, Universitaet Karlsruhe (TH), Fakultaet fuer Wirtschaftwsissenschaften.

[Agarwal & Studer, 2006] Agarwal, S. & Studer, R. (2006). Automatic matchmaking of web services. In *Proceedings of IEEE 2006 the International Conference on Web Services (ICWS2006)* (pp. 45–54).

[Alonso et al., 2003] Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2003). *Web Services, Concepts, Architecture and Applications*. Number ISBN 3-540-44008-9. Tiergartenstrasse 17, D-69121 Heidelberg, Germany: Springer Verlag, 1 edition.

[Alves et al., 2007] Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guizar, A., Kartha, N., Liu, C. K., Khalaf, R., Koenig, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., & Yiu, A. (2007). Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. OASIS standard.

[Arkin et al., 2002] Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., Pogliani, S., Riemer, K., Struble, S., Takacsi-Nagy, P., Trickovic, I., & Zimek, S. (2002). Web Service Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/wsci/. W3C Note.

[Arroyo et al., 2005] Arroyo, S., Cimpian, E., Domingue, J., Feier, C., Fensel, D., Koenig-Ries, B., Lausen, H., Polleres, A., & Stollberg, M. (2005). Web Service Modeling Ontology Primer. http://www.w3.org/Submission/WSMO-primer/. W3C Member Submission.

[Austin et al., 2004] Austin, D., Barbir, A., Peters, E., & Ross-Talbot, S. (2004). Web Services Choreography Requirements. http://www.w3.org/TR/ws-chor-reqs/. W3C Working Draft.

[Baader et al., 2007] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P., Horrocks, I., Moeller, R., Haarslev, V., & Sattler, U. (2007). *The Description Logic Handbook: Theory, Implementation and Applications*. Number ISBN 0-521-87625-7. Edinburgh Building, Shaftesbury Road, Cambridge, CB2 8RU, UK: Cambridge University Press, 2 edition.

[Baader & Sattler, 2001] Baader, F. & Sattler, U. (2001). An overview of tableau algorithms for description logics. *Studia Logica*, 69, 5–40.

[Baier et al., 2006] Baier, J. A., Hussell, J., Bacchus, F., & McIlraith, S. A. (2006). Planning with temporally extended preferences by heuristic search. In *Proceedings of the ICAPS06 Workshop on Planning with Preferences* (pp. 7–10). Lake District, UK. A version of this paper also appeared in the Fifth International Planning Competition (IPC-5) Booklet.

[Baier & McIlraith, 2006a] Baier, J. A. & McIlraith, S. A. (2006a). On planning with programs that sense. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR2006)* (pp. 492–502). Lake District, UK.

[Baier & McIlraith, 2006b] Baier, J. A. & McIlraith, S. A. (2006b). Planning with temporally extended goals using heuristic search. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS06)* (pp. 342–345). Lake District, UK.

[Baier & Pinto, 1999] Baier, J. A. & Pinto, J. A. (1999). Integrating true concurrency into the robot programming language. In *Proceedings of the 9th International Conference of the Chilean Computer Society (SCCC 1999), Talca, Chile* (pp. 179–186).: IEEE Computer Society.

[Banerji et al., 2002] Banerji, A., Bartolini, C., Beringer, D., Chopella, V., Govindarajan, K., Karp, A., Kuno, H., Lemon, M., Pogossiants, G., Sharma, S., & Williams, S. (2002). Web Services Conversation Language (WSCL) 1.0. http://www.w3.org/TR/wscl10/. W3C Note.

[Barreto et al., 2007] Barreto, C., Bullard, V., Erl, T., Evdemon, J., Jordan, D., Kand, K., Koenig, D., Moser, S., Stout, R., Ten-Hove, R., Trickovic, I., van der Rijn, D., & Yiu, A. (2007). Web Services Business Process Execution Language Version 2.0: Primer. http://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm.

[Battle et al., 2005a] Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., McIlraith, S., McGuinness, D., Su, J., & Tabet, S. (2005a). Semantic Web Services Language (SWSL). http://www.w3.org/Submission/SWSF-SWSL/. W3C Member Submission.

[Battle et al., 2005b] Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., McIlraith, S., McGuinness, D., Su, J., & Tabet, S. (2005b). Semantic Web Services Ontology (SWSO). http://www.w3.org/Submission/SWSF-SWSO/. W3C Member Submission.

[Battle et al., 2005c] Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., McIlraith, S. A., McGuinness, D., Su, J., & Tabet, S. (2005c). Semantic Web Services Framework (SWSF) Overview. http://www.w3.org/Submission/SWSF/. W3C Member Submission.

[Bechhofer et al., 2004] Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., & Stein, L. A. (2004). OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/. W3C Recommendation.

[Ben-Ari et al., 1982] Ben-Ari, M., Halpern, J. Y., & Pnueli, A. (1982). Deterministic propositional dynamic logic: Finite models, complexity, and completeness. *Journal of Computer and System Science*, 25(3), 402–417.

[Berardi et al., 2005a] Berardi, D., Calvanese, D., Giacomo, G. D., Hull, R., & Mecella, M. (2005a). Automatic composition of transition-based semantic web services with messaging. In *Proceedings of the 31st International Conference on Very Large Databases (VLDB 2005), Trondheim, Norway* (pp. 613–624).

[Berardi et al., 2005b] Berardi, D., Calvanese, D., Giacomo, G. D., Hull, R., & Mecella, M. (2005b). Towards automatic web service discovery and composition in a context with semantics, messages and internal process flow (a position paper). In *Proceedings of the W3C Workshop on Frameworks for Semantics in Web Services (SWSF 2005)*.

[Berardi et al., 2003a] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., & Mecella, M. (2003a). Automatic composition of e-services that export their behavior. In *Proceedings of the 1st International Conference on Service-Oriented Computing (ICSOC2003), Trento, Italy* (pp. 43–58).

[Berardi et al., 2003b] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., & Mecella, M. (2003b). e-service composition by description logics based reasoning. In *Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rome, Italy*.

[Berardi et al., 2003c] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., & Mecella, M. (2003c). *A Foundational Framework for e-Services*. Technical report, Dipartimento di Informatica e Sistemistica Università di Roma La Sapienza, Via Salaria 113, 00198 Roma, Italy.

[Berardi et al., 2004a] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., & Mecella, M. (2004a). ESC: A tool for automatic composition of e-services based on logics of programs. In *Proceedings of the 5th International Workshop on Technologies for E-Services (TES 2004), revised selected papers, Toronto, Canada* (pp. 80–94).

[Berardi et al., 2004b] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., & Mecella, M. (2004b). Synthesis of composite e-services based on automated reasoning. In *Proceedings of the ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services (P4WGS2004)*.

[Berardi et al., 2005c] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., & Mecella, M. (2005c). Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems (IJCIS)*, 14(4), 333–376.

[Berardi et al., 2005d] Berardi, D., Calvanese, D., Giacomo, G. D., & Mecella, M. (2005d). Composition of services with nondeterministic observable behavior. In *Proceedings of the 3rd International Conferenceon on Service-Oriented Computing (ICSOC2005), Amsterdam, The Netherlands* (pp. 520–526).

[Berardi et al., 2008] Berardi, D., Cheikh, F., Giacomo, G. D., & Patrizi, F. (2008). Automatic service composition via simulation. *International Journal of Foundations of Computer Science (IJFCS)*, 19(2), 429–451.

[Berardi et al., 2004c] Berardi, D., Giacomo, G. D., Lenzerini, M., Mecella, M., & Calvanese, D. (2004c). Synthesis of underspecified composite e-services based on automated reasoning. In M. Aiello, M. Aoyama, F. Curbera, & M. P. Papazoglou (Eds.), *Proceedings of the 2nd International Conference on Service-Oriented Computing (ICSOC2004), New York, NY, USA* (pp. 105–114).: ACM.

[Berardi et al., 2006a] Berardi, D., Giacomo, G. D., Mecella, M., & Calvanese, D. (2006a). *Automatic Composition of Web Services with Nondeterministic Behavior*. Technical report, Dipartimento di Informatica e Sistemistica Universita di Roma La Sapienza and Libera Universit a di Bolzano/Bozen Facolta di Scienze e Tecnologie Informatiche, Via Salaria 113, 00198 Roma, Italy and Piazza Domenicani 3, 39100 Bolzano, Italy.

[Berardi et al., 2006b] Berardi, D., Giacomo, G. D., Mecella, M., & Calvanese, D. (2006b). Composing web services with nondeterministic behavior. In *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS2006), Chicago, Illinois, USA* (pp. 909–912).

[Bray et al., 2008] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/xml/. W3C Recommendation.

[Bultan et al., 2003] Bultan, T., Fu, X., Hull, R., & Su, J. (2003). Conversation specification: a new approach to design and analysis of e-service composition. In *Proceedings of the 12th International World Wide Web Conference (WWW2003), Budapest, Hungary* (pp. 403–410).: ACM Press.

[Burdett & Kavantzas, 2004] Burdett, D. & Kavantzas, N. (2004). Web Service Choreography Model Overview. http://www.w3.org/TR/ws-chor-model/. W3C Working Draft.

[Bussler et al., 2005] Bussler, C., Cimpian, E., Fensel, D., Gomez, J. M., Haller, A., Haselwanter, T., Kerrigan, M., Mocan, A., Moran, M., Oren, E., Sapkota, B., Toma, I., Viskova, J., Vitvar, T., Zaremba, M., & Zaremba, M. (2005). Web Service Execution Environment (WSMX). http://www.w3.org/Submission/WSMX/. W3C Member Submission.

[Calvanese et al., 2007] Calvanese, D., Giacomo, G., Lembo, D., Lenzerini, M., & Rosati, R. (2007). Tractable reasoning and efficient query answering in description logics: The DL-lite family. *Journal of Automated Reasoning*, 39(3), 385–429.

[Calvanese et al., 2006] Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M., & Rosati, R. (2006). Data complexity of query answering in description logics. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning, Lake District, UK* (pp. 260–270).

[Calvanese et al., 2008a] Calvanese, D., Giacomo, G. D., & Lenzerini, M. (2008a). Conjunctive query containment and answering under description logic constraints. *ACM Transactions on Computational Logic (TOCL)*, 9(3), 22:1–22:3.

[Calvanese et al., 2008b] Calvanese, D., Giacomo, G. D., Lenzerini, M., Mecella, M., & Patrizi, F. (2008b). Automatic service composition and synthesis: the roman model. *IEEE Data Engineering Bulletin*, 31(3), 18–22.

[Cardoso, 2007] Cardoso, J. (2007). *Semantic Web Services, Theory, Tools and Applications*. Number ISBN 159904045X. 701 E. Chocolate Avenue, Hershey, PA 17033, USA: IGI Global Publishing, 1 edition.

[Castagna, 1994] Castagna, G. (1994). Covariance and contravariance conflict without a cause.

[Chinnici et al., 2007a] Chinnici, R., Haas, H., Lewis, A. A., Moreau, J.-J., Orchard, D., & Weerawarana, S. (2007a). Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts. http://www.w3.org/TR/wsdl20-adjuncts/.

[Chinnici et al., 2007b] Chinnici, R., Moreau, J.-J., Ryman, A., & Weerawarana, S. (2007b). Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. http://www.w3.org/TR/wsdl20/.

[Colucci et al., 2003a] Colucci, S., Noia, T. D., Sciascio, E. D., Donini, F. M., & Mongiello, M. (2003a). Concept abduction and contraction in description logics. In *Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rome, Italy*.

[Colucci et al., 2003b] Colucci, S., Noia, T. D., Sciascio, E. D., Donini, F. M., & Mongiello, M. (2003b). Description logics approach to semantic matching of web services. *Journal of Computing and Information Technology*, 11(3), 217–224. Invited paper.

[Colucci et al., 2005a] Colucci, S., Noia, T. D., Sciascio, E. D., Donini, F. M., & Mongiello, M. (2005a). Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an e-marketplace. *Electronic Commerce Research and Applications*, 4(4), 345–361.

[Colucci et al., 2004] Colucci, S., Noia, T. D., Sciascio, E. D., Donini, F. M., Mongiello, M., Piscitelli, G., & Rossi, G. (2004). An agency for semantic-based automatic discovery of web services. In *Proceedings of Artificial Intelligence Applications and Innovations (AIAI2004)*: Kluwer.

[Colucci et al., 2005b] Colucci, S., Noia, T. D., Sciascio, E. D., Donini, F. M., & Ragone, A. (2005b). Automated task-oriented team composition using description logics. In *Proceedings of 5th International Conference on Knowledge Management (I-KNOW2005), Graz, Austria*.

[de Bruijn, 2008] de Bruijn, J. (2008). *WSML Abstract Syntax and Semantics*. Final Draft d16.3, WSML Working Group.

[de Bruijn et al., 2005a] de Bruijn, J., Bussler, C., Domingue, J., Fensel, D., Hepp, M., Keller, U., Kifer, M., Koenig-Ries, B., Kopecky, J., Lausen, H., Oren, E., Polleres, A., Roman, D., Scicluna, J., & Stollberg, M. (2005a). Web Service Modeling Ontology (WSMO). http://www.w3.org/Submission/WSMO/. W3C Member Submission.

[de Bruijn et al., 2005b] de Bruijn, J., Fensel, D., Keller, U., Kifer, M., Lausen, H., Krummenacher, R., Polleres, A., & Predoiu, L. (2005b). Web Service Modeling Language (WSML). http://www.w3.org/Submission/WSML/. W3C Member Submission.

[de Bruijn et al., 2005c] de Bruijn, J., Fensel, D., Kifer, M., Kopeck, J., Lausen, H., Polleres, A., Roman, D., Scicluna, J., & Toma, I. (2005c). Relationship of WSMO to Other Relevant Technologies. http://www.w3.org/Submission/WSMO-related/. W3C Member Submission.

[de Bruijn & Heymans, 2007] de Bruijn, J. & Heymans, S. (2007). A semantic framework for language layering in WSML. In *Proceedings of the 1st International Conference on Web Reasoning and Rule Systems (RR2007)* (pp. 103–117). Innsbruck, Austria: Springer.

[de Bruijn et al., 2008] de Bruijn, J., Kopecky, J., Toma, I., Steinmetz, N., Foxvog, D., Keller, U., Kerrigan, M., Krummenacher, R., Lausen, H., Sirbu, A., Roman, D., Scicluna, J., Fensel, D., & Kifer, M. (2008). WSML/RDF. http://www.wsmo.org/TR/d32/v1.0/. WSML Final Draft D32v1.0.

[de Bruijn et al., 2005d] de Bruijn, J., Lausen, H., Polleres, A., & Fensel, D. (2005d). The WSML rule languages for the semantic web. In *Proceedings of the W3C Workshop on Rule Languages for Interoperability* Washington DC, USA. Position paper.

[Deutsch et al., 2009] Deutsch, A., Hull, R., Patrizi, F., & Vianu, V. (2009). Automatic verification of data-centric business processes. In R. Fagin (Ed.), *Proceedings of the 12th International Conference on Database Theory (ICDT), St. Petersburg, Russia*, volume 361 of *ACM International Conference Proceeding Series* (pp. 252–267).: ACM Press.

[Deutsch et al., 2004] Deutsch, A., Sui, L., & Vianu, V. (2004). Specification and verification of data-driven web services. In A. Deutsch (Ed.), *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems PODS 2004, Paris, France*, number ISBN 1-58113-858-X (pp. 71–82).

[Deutsch et al., 2007] Deutsch, A., Sui, L., & Vianu, V. (2007). Specification and verification of data-driven web applications. *Journal of Computer and System Sciences (JCSS)*, 73(3), 442–474.

[Deutsch et al., 2006a] Deutsch, A., Sui, L., Vianu, V., & Zhou, D. (2006a). A system for specification and verification of interactive, data-driven web applications. In S. Chaudhuri, V. Hristidis, & N. Polyzotis (Eds.), *Proceedings of the ACM SIGMOD 2006 International Conference on Management of Data, Chicago, Illinois, USA*, number ISBN 1-59593-256-9 (pp. 772–774).: ACM Press.

[Deutsch et al., 2006b] Deutsch, A., Sui, L., Vianu, V., & Zhou, D. (2006b). Verification of communicating data-driven web services. In S. Vansummeren (Ed.), *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems PODS 2006, Chicago, Illinois, USA*, number ISBN 1-59593-318-2 (pp. 90–99).: ACM Press.

[Domingue et al., 2008] Domingue, J., Fensel, D., & Gonzalez-Cabero, R. (2008). Soa4all, enabling the soa revolution on a world wide scale. *International Conference on Semantic Computing*, 0, 530–537.

[Erl, 2005] Erl, T. (2005). *Service-Oriented Architecture (SOA) Concepts, Technology, and Design*. Number ISBN 0-131-85858-0 in Prentice Hall PTR Service-Oriented Computing Series. Upper Saddle River, NJ 07458, USA: Prentice Hall, Pearson Education, Inc., 1 edition.

[Erl, 2007] Erl, T. (2007). *SOA Principles of Service Design*. Number ISBN 0-132-34482-3 in Prentice Hall PTR Service-Oriented Computing Series. Upper Saddle River, NJ 07458, USA: Prentice Hall, Pearson Education, Inc., 1 edition.

[Erl, 2009] Erl, T. (2009). *SOA Design Patterns*. Number ISBN 0-136-13516-1 in Prentice Hall PTR Service-Oriented Computing Series. Upper Saddle River, NJ 07458, USA: Prentice Hall, Pearson Education, Inc., 1 edition.

[Fadel & McIlraith, 2002] Fadel, R. & McIlraith, S. A. (2002). Planning with complex actions. In *Proceedings of the Ninth International Workshop on Non-Monotonic Reasoning (NMR2002) Toulouse, France* (pp. 356–364). Toulouse, France.

[Fagin et al., 2003] Fagin, R., Halpern, J. Y., Moses, Y., & Vardi, M. Y. (2003). *Reasoning About Knowledge*. Number ISBN 0-262-56200-6. 292 Main Street, Cambridge, MA 02142, USA: The MIT Press, 1 edition.

[Fischer & Ladner, 1979] Fischer, M. J. & Ladner, R. E. (1979). Propositional dynamic logic of regular programs. *Journal of Computer and System Science*, 18(2), 194–211.

[Gerede et al., 2004] Gerede, C. E., Hull, R., Ibarra, O. H., & Su, J. (2004). Automated composition of e-services: Lookaheads. In M. Aiello, M. Aoyama, F. Curbera, & M. P. Papazoglou (Eds.), *Proceedings of the 2nd International Conference on Service-Oriented Computing (ICSOC2004), New York, NY, USA* (pp. 252–262).

[Ghandeharizadeh et al., 2003] Ghandeharizadeh, S., Knoblock, C. A., Papadopoulos, C., Shahabi, C., Alwagait, E., Ambite, J. L., Cai, M., Chen, C.-C., Pol, P., Schmidt, R. R., Song, S., Thakkar, S., & Zhou, R. (2003). Proteus: A system for dynamically composing and intelligently executing web services. In *Proceedings of the International Conference on Web Services (ICWS2003), Las Vegas, Nevada, USA* (pp. 17–21).

[Giacomo et al., 1997] Giacomo, G. D., Lesperance, Y., & Levesque, H. J. (1997). Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In M. Pollack (Ed.), *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI1997), Nagoya, Japan* (pp. 1221–1226). San Francisco: Morgan Kaufmann Publishers.

[Giacomo et al., 2000] Giacomo, G. D., Lesperance, Y., & Levesque, H. J. (2000). ConGolog, a concurrent programming language based on the situation calculus. *Journal of Artificial Intelligence*, 121(1-2), 109–169.

[Giacomo et al., 2009] Giacomo, G. D., Lesperance, Y., Levesque, H. J., & Sardina, S. (2009). *IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents*, chapter 1, (pp. 389). Number ISBN 978-0-387-89298-6. Springer Verlag.

[Giacomo & Massacci, 1996] Giacomo, G. D. & Massacci, F. (1996). Tableaux and algorithms for propositional dynamic logic with converse. In *Proceedings of the 13th International Conference on Automated Deduction (CADE1996)*, volume 1104 of *Lecture Notes in Computer Science* (pp. 613–628).: Springer Verlag.

[Giacomo & Massacci, 2000] Giacomo, G. D. & Massacci, F. (2000). Combining deduction and model checking into tableaux and algorithms for converse-PDL. *Elsevier Journal of Information and Computation*, 160, 117–137.

[Giacomo & Sardina, 2009] Giacomo, G. D. & Sardina, S. (2009). Composition of ConGolog programs. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI2009), Pasadena, California, USA*.

[Grimm & Hitzler, 2008] Grimm, S. & Hitzler, P. (2008). Semantic matchmaking of web resources with local closed-world reasoning. *International Journal of e-Commerce*, 12(2), 89–126.

[Grimm et al., 2004] Grimm, S., Motik, B., & Preist, C. (2004). Variance in e-business service discovery. In D. Martin, R. Lara, & T. Yamaguchi (Eds.), *Proceedings of the 3rd International Semantic Web*

Conference (ISWC2004), Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications, Hiroshima, Japan, volume 119 of *CEUR Workshop Proceedings* Hiroshima, Japan.

[Grimm et al., 2006] Grimm, S., Motik, B., & Preist, C. (2006). Matching semantic service descriptions with local closed-world reasoning. In Y. Sure & J. Domingue (Eds.), *Proceedings of the 3rd European Semantic Web Conference (ESWC 2006), The Semantic Web: Research and Applications, Budva, Montenegro*, volume 4011 of *Lecture Notes in Computer Science* (pp. 575–589).: Springer Verlag.

[Grosof et al., 2003] Grosof, B. N., Horrocks, I., Volz, R., & Decker, S. (2003). Description logic programs: Combining logic programs with description logic. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*, number ISBN 1-58113-680-3 (pp. 48–57).: ACM Press.

[Grüninger et al., 2008] Grüninger, M., Hull, R., & McIlraith, S. A. (2008). A short overview of flows: A first-order logic ontology for web services. *IEEE Data Engeering Bulletin*, 31(3), 3–7.

[Gu & Soutchanski, 2006] Gu, Y. & Soutchanski, M. (2006). The two-variable situation calculus. In *Proceedings of the 3rd European Starting AI Researcher Symposium (STAIRS-06) at ECAI06, Riva del Garda, Italy* (pp. 144–161).

[Haarslev & Möller, 2001a] Haarslev, V. & Möller, R. (2001a). Description of the racer system and its applications. In *Working Notes of the 2001 International Description Logics Workshop (DL2001), Stanford, CA, USA*.

[Haarslev & Möller, 2001b] Haarslev, V. & Möller, R. (2001b). Racer system description. In *Proceedings of the 1st International Joint Conference of Automated Reasoning (IJCAR 2001), Siena, Italy* (pp. 701–706).

[Haarslev & Möller, 2003] Haarslev, V. & Möller, R. (2003). Racer: A core inference engine for the semantic web. In *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003) at the 2nd International Semantic Web Conference (ISWC 2003), Sundial Resort, Sanibel Island, Florida, USA*.

[Haas, 1987] Haas, A. R. (1987). The case for domain-specific frame axioms. In *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*: Morgan Kaufmann Publishers Inc.

[Hoare, 1969] Hoare, S. C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 576–580.

[Hobbs & Pan, 2004] Hobbs, J. R. & Pan, F. (2004). An ontology of time for the semantic web. *ACM Transactions on Asian Language Information Processing (TALIP)*, 3(1), 66–85.

[Hobbs & Pan, 2006] Hobbs, J. R. & Pan, F. (2006). *Time Ontology in OWL*. World wide web consortium (w3c) working draft, W3C World Wide Web Consortium. http://www.w3.org/TR/2006/WD-owl-time-20060927/.

[Hobbs & Pustejovsky, 2003] Hobbs, J. R. & Pustejovsky, J. (2003). Annotating and reasoning about time and events. In P. Doherty, J. McCarthy, & M.-A. Williams (Eds.), *Working Papers of the AAAI 2003 Spring Symposium on Logical Formalization of Commonsense Reasoning* (pp. 74–82). University of Southern California, Menlo Park, California: AAAI Press.

[Hull et al., 2006] Hull, D., Horrocks, I., Zolin, E., Bovykin, A., Sattler, U., & Stevens, R. (2006). Deciding semantic matching of stateless services. In *Proceedings of the 21st International Conference on Artificial Intelligence (AAAI-06)* (pp. 1319–1324).

[Hull, 2005] Hull, R. (2005). Web services composition: A story of models, automata, and logics. In *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005), Orlando, FL, USA*.

[Hull et al., 2003] Hull, R., Benedikt, M., Christophides, V., & Su, J. (2003). E-services: a look behind the curtain. In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART 2003 Symposium on Principles of Database Systems, San Diego, CA, USA* (pp. 1–14).

[Hustadt et al., 2005] Hustadt, U., Motik, B., & Sattler, U. (2005). Data complexity of reasoning in very expressive description logics. In *Proceedings of 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)* (pp. 466–471).: Morgan Kaufmann Publishers.

[Kashyap et al., 2008] Kashyap, V., Moran, M., & Bussler, C. (2008). *The Semantic Web, Semantics for Data and Services on the Web*. Number ISBN 978-3-540-76451-9 in Data-Centric Systems and Applications. Tiergartenstrasse 17, D-69121 Heidelberg, Germany: Springer Verlag, 1 edition.

[Kavantzas et al., 2005] Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., & Barreto, C. (2005). Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/ws-cdl-10/. W3C Candidate Recommendation.

[Kifer et al., 1995] Kifer, M., Lausen, G., & Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the Association for Computing Machinery (ACM)*, 42(4), 741–843.

[Knoblock et al., 2005] Knoblock, C. A., Thakkar, S., & Ambite, J. L. (2005). Composing, optimizing, and executing plans for bioinformatics web services. *VLDB Journal, Special Issue on Data Management, Analysis and Mining for Life Sciences*, 14(3), 330–353.

[Lausen et al., 2006] Lausen, H., de Bruijn, J., Keller, U., & Lara, R. (2006). Semantic web services with wsmo. *Upgrade, European Journal for the Informatics Professional, Special issue on Web Services*, VII(5), 34–37.

[Lausen et al., 2005] Lausen, H., de Bruijn, J., Polleres, A., & Fensel, D. (2005). WSML - a language framework for semantic web services. In *Proceedings of the W3C Workshop on Rule Languages for Interoperability* Washington DC, USA. Position paper.

[Lécué & Delteil, 2007] Lécué, F. & Delteil, A. (2007). Making the difference in semantic web service composition. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI2007), Vancouver, British Columbia, Canada* (pp. 1383–1388).

[Lécué & Leger, 2006] Lécué, F. & Leger, A. (2006). A formal model for semantic web service composition. In I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, & L. Aroyo (Eds.), *Proceedings of the 5th International Semantic Web Conference (ISWC 2006), Athens, GA, USA* (pp. 385–398).

[Lesperance et al., 2008] Lesperance, Y., Giacomo, G. D., & Ozgovde, A. N. (2008). A model of contingent planning for agent programming languages. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal* (pp. 477–484).

[Lesperance et al., 2000] Lesperance, Y., Levesque, H. J., Lin, F., & Scherl, R. B. (2000). Ability and knowing how in the situation calculus. *Studia Logica*, 66(1), 165–186.

[Lesperance et al., 1999] Lesperance, Y., Levesque, H. J., & Reiter, R. (1999). A situation calculus approach to modeling and programming agents. In M. J. Wooldridge & A. Rao (Eds.), *Foundations of Rational Agency* (pp. 275–299). Dordrecht: Kluwer Academic Publishers.

[Levesque & Brachman, 2004] Levesque, H. J. & Brachman, R. (2004). *Knowledge Representation and Reasoning*. Number ISBN 1-558-60932-6 in The Morgan Kaufmann Series in Artificial Intelligence. 340 Pine St, 6th floor. San Fransisco, CA 94104, USA: Morgan Kaufmann Publishers Inc.

[Levesque & Lakemeyer, 2001] Levesque, H. J. & Lakemeyer, G. (2001). *The Logic of Knowledge Bases*. Number ISBN 0-262-12232-4. 292 Main Street, Cambridge, MA 02142, USA: The MIT Press.

[Levesque et al., 1998] Levesque, H. J., Pirri, F., & Reiter, R. (1998). Foundations for the situation calculus. *Electronic Transaction on Artificial Intelligence (ETAI)*, 2, 159–178.

[Levesque et al., 1997] Levesque, H. J., Reiter, R., Lesperance, Y., Lin, F., & Scherl, R. B. (1997). GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3), 59–83.

[Li, 2004] Li, L. (2004). A software framework for matchmaking based on semantic web technology. *International Journal of Electronic Commerce*, 8(4), 1.

[Li & Horrocks, 2003a] Li, L. & Horrocks, I. (2003a). Matchmaking using an instance store: Some preliminary results. In *Proceedings of the 2003 International Workshop on Description Logics (DL2003), poster paper*.

[Li & Horrocks, 2003b] Li, L. & Horrocks, I. (2003b). A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003), Budapest, Hungary* (pp. 331–339).: ACM.

[Lin & Reiter, 1997] Lin, F. & Reiter, R. (1997). Rules as actions: A situation calculus semantics for logic programs. *Journal of Logic Programming*, 31(1-3), 299–330.

[Lynch, 1997] Lynch, N. A. (1997). *Distributed Algorithms*. Number ISBN 1558603484 in Morgan Kaufmann Series in Data Management Systems. 340 Pine St, 6th floor. San Fransisco, CA 94104, USA: Morgan Kaufmann Publishers Inc., 1 edition.

[Manna & Waldinger, 1980] Manna, Z. & Waldinger, R. J. (1980). A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1), 90–121.

[Martin et al., 2004a] Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S. A., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., & Sycara, K. (2004a). OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/. W3C Member Submission.

[Martin et al., 2007] Martin, D. H., McIlraith, S. A., Burstein, M., McDermott, D., Paolucci, M., Sycara, K., McGuinness, D. L., Sirin, E., & Srinivasan, N. (2007). Bringing semantics to web services with OWL-S. *World Wide Web Journal*, 10(3), 243–277. Special Issue: Recent Advances in Web Services.

[Martin et al., 2004b] Martin, D. H., Paolucci, M., McIlraith, S. A., Burstein, M., McDermott, D., McGuinness, D. L., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., & Sycara, K. (2004b). Bringing semantics to web services: The OWL-S approach. In *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC2004)* (pp. 26–42). San Diego, CA, USA. Revised Selected Papers.

[McCarthy, 1963] McCarthy, J. (1963). *Situations, Actions and Causal Laws*. Stanford University Artificial Intelligence Project 13, Department of Computer Science, Stanford University, Stanford, CA 94305, USA. Accession number: AD0785031.

[McCarthy, 2001a] McCarthy, J. (2001a). Actions and other events in situation calculus. In *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR2002), Toulouse, France*.

[McCarthy, 2001b] McCarthy, J. (2001b). Situation calculus with concurrent events and narrative. *Technical Report of Stanford*, 1, 1.

[McCarthy & Hayes, 1969] McCarthy, J. & Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 4* (pp. 463–502). Edinburgh University Press. reprinted in McC90.

[McGuinness & van Harmelen, 2004] McGuinness, D. L. & van Harmelen, F. (2004). OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/. W3C Recommendation.

[McIlraith, 1999] McIlraith, S. A. (1999). Model-based programming using golog and the situation calculus. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis (DX'99)* (pp. 184–192). Loch Awe Hotel, Scotland, UK.

[McIlraith & Son, 2001] McIlraith, S. A. & Son, T. C. (2001). Adapting golog for programming the semantic web. In *Proceedings of the 5th Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense 2001)* (pp. 195–202). Warren Weaver Hall, room 109, 251 Mercer St., between 3rd and 4th Streets, New York City, NY, USA.

[McIlraith & Son, 2002] McIlraith, S. A. & Son, T. C. (2002). Adapting golog for composition of semantic web services. In *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR2002), Toulouse, France* (pp. 482–493). Toulouse, France.

[McIlraith et al., 2001] McIlraith, S. A., Son, T. C., & Zeng, H. (2001). Semantic web services. *IEEE Intelligent Systems. Special Issue on the Semantic Web*, 16(2), 46–53.

[Mecella et al., 2004] Mecella, M., Berardi, D., Rosa, F. D., & Santis, L. D. (2004). Finite state automata as conceptual model for E-services. Journal of Integrated Design and Process Science.

[Meyer, 1990] Meyer, B. (1990). *Introduction to the Theory of Programming Languages*. Number ISBN 01-3498510-9 in Prentice-Hall International Series in Computer Science. Prentice Hall, Pearson Education, Inc., 1st edition.

[Milner, 1971] Milner, A. J. R. G. (1971). An algebraic definition of simulation between programs. In D. C. Cooper (Ed.), *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI 1971), London, UK*, number ISBN 0-934613-34-6 (pp. 481–489).: William Kaufmann.

[Milner, 1999] Milner, R. (1999). *Communicating and Mobile Systems: the Pi-Calculus*. Number ISBN 0-521-65869-1. Cambridge University Press, 1 edition.

[Möller & Haarslev, 2003] Möller, R. & Haarslev, V. (2003). Description logics for the semantic web: Racer as a basis for building agent systems. *Zeitschrift der Künstliche Intelligenz (KI)*, 17(3), 10ff.

[Muscholl & Walukiewicz, 2007] Muscholl, A. & Walukiewicz, I. (2007). A lower bound on web services composition. In *Proceedings of the 10th International Conference on Foundations of Software Science and Computational Structures FOSSACS 2007, Braga, Portugal* (pp. 274–286).

[Muscholl & Walukiewicz, 2008] Muscholl, A. & Walukiewicz, I. (2008). A lower bound on web services composition. *Journal of Logical Methods in Computer Science*, abs/0804.3105, 1–14.

[Narayanan & McIlraith, 2002] Narayanan, S. & McIlraith, S. A. (2002). Simulation, verification and automated composition of web services. In *Proceedings of the 11th International World Wide Web Conference (WWW-11)* (pp. 77–88). Honolulu, Hawaii, USA.

[Narayanan & McIlraith, 2003] Narayanan, S. & McIlraith, S. A. (2003). Analysis and simulation of web services. *Computer Networks*, 42(5), 675–693.

[Noia et al., 2007] Noia, T. D., Sciascio, E. D., & Donini, F. M. (2007). Semantic matchmaking as non-monotonic reasoning: A description logic approach. *Journal of Artificial Intelligence Research (JAIR)*, 29, 269–307.

[Noia et al., 2008] Noia, T. D., Sciascio, E. D., & Donini, F. M. (2008). A nonmonotonic approach to semantic matchmaking and request refinement in e-marketplaces. *International Journal of Electronic Commerce (IJEC)*, 12(2), 127–154.

[Noia et al., 2005] Noia, T. D., Sciascio, E. D., Donini, F. M., Ragone, A., & Colucci, S. (2005). Automated semantic web services orchestration via concept covering. In *Proceedings of the 14th International Conference on World Wide Web (WWW2005), Chiba, Japan* (pp. 1160–1161).: ACM Press.

[Ortiz et al., 2006a] Ortiz, M., Calvanese, D., & Eiter, T. (2006a). Characterizing data complexity for conjunctive query answering in expressive description logics. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)* (pp. 275–280).

[Ortiz et al., 2006b] Ortiz, M., Calvanese, D., & Eiter, T. (2006b). Data complexity of answering unions of conjunctive queries in SHIQ. In *Proceedings of the 2006 International Workshop on Description Logics (DL2006), Lake District, UK*, volume 189.

[Ortiz et al., 2008] Ortiz, M., Calvanese, D., & Eiter, T. (2008). Data complexity of query answering in expressive description logics via tableaux. *Journal of Automated Reasoning*, 41(1), 61–98.

[Pan, 2007] Pan, F. (2007). *Representing Complex Temporal Phenomena for the Semantic Web and Natural Language*. Phd thesis, Faculty of the Graduate School, University of Southern California, Los Angeles, USA.

[Pan & Hobbs, 2005] Pan, F. & Hobbs, J. R. (2005). Temporal aggregates in OWL-time. In I. Russell & Z. Markov (Eds.), *Proceedings of the 8th International Florida Artificial Intelligence Research Society Conference, Clearwater Beach, Florida, USA*, number ISBN 1-57735-234-3 (pp. 560–565).: AAAI Press.

[Pan et al., 2006] Pan, F., Mulkar-Mehta, R., & Hobbs, J. R. (2006). Learning event durations from event descriptions. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (ACL 2006), Sydney, Australia*.

[Paolucci et al., 2002] Paolucci, M., Kawamura, T., Payne, T. R., & Sycara, K. P. (2002). Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference (ISWC 2002), Sardinia, Italy*, volume 2342 of *Lecture Notes in Computer Science* (pp. 333–347).: Springer Verlag.

[Patel-Schneider et al., 2004] Patel-Schneider, P. F., Hayes, P., & Horrocks, I. (2004). OWL Web Ontology Language Semantics and Abstract Syntax. http://www.w3.org/TR/owl-semantics/. W3C Recommendation.

[Patrizi & Giacomo, 2009] Patrizi, F. & Giacomo, G. D. (2009). Composition of services that share an infinite-state blackboard. In *Proceedings of IJCAI-2009 Workshop on Information Integration on the Web (IIWEB2009)*.

[Perrin et al., 1990] Perrin, D., Berstel, J., Boasson, L., Salomaa, A., Thomas, W., Courcelle, B., Dershowitz, N., Jouannaud, J. P., Barendregt, H., Mitchell, J., Courcelle, B., Apt, K., Mosses, P., Gunter, G., & Scott, D. (1990). *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Number ISBN 0-444-88074-7. Elsevier Science.

[Pinto, 1994] Pinto, J. A. (1994). *Temporal Reasoning in the Situation Calculus*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada.

[Pinto & Reiter, 1993] Pinto, J. A. & Reiter, R. (1993). Temporal reasoning in logic programming: A case for the situation calculus. In D. S. Warren (Ed.), *Proceedings of the 10th International Conference on Logic Programming* (pp. 203–221). Budapest, Hungary: The MIT Press.

[Pinto & Reiter, 1995] Pinto, J. A. & Reiter, R. (1995). Reasoning about time in the situation calculus. *Annal of Mathematics and Artificial Intelligence*, 14(2-4), 251–268.

[Pirri & Reiter, 1999] Pirri, F. & Reiter, R. (1999). Some contributions to the metatheory of the situation calculus. *ACM Transactions on Computational Logics*, 46(3), 325–361.

[Pirri & Reiter, 2000] Pirri, F. & Reiter, R. (2000). Planning with natural actions in the situation calculus. In J. Minker (Ed.), *Logic-Based Artificial Intelligence* (pp. 213–231). Norwell, MA, USA: Kluwer Academic Publishers.

[Pistore et al., 2004] Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., & Traverso, P. (2004). Planning and monitoring web service composition. In D. F. Christoph Bussler (Ed.), *Proceedings of the 11th International Conference, Artificial Intelligence: Methodology, Systems, and Applications, AIMSA 2004, Varna, Bulgaria*, volume 3192 of *Lecture Notes in Computer Science*: Springer.

[Pistore et al., 2005a] Pistore, M., Marconi, A., Bertoli, P., & Traverso, P. (2005a). Automated composition of web services by planning at the knowledge level. In L. P. Kaelbling & A. Saffiotti (Eds.), *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, UK*, number ISBN 0938075934 (pp. 1252–1259).: Professional Book Center.

[Pistore et al., 2005b] Pistore, M., Traverso, P., & Bertoli, P. (2005b). Automated composition of web services by planning in asynchronous domains. In S. Biundo, K. L. Myers, & K. Rajan (Eds.), *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS2005), Monterey, California, USA*, number ISBN 1-57735-220-3 (pp. 2–11).: AAAI.

[Ragone et al., 2007] Ragone, A., Noia, T. D., Sciascio, E. D., Donini, F. M., Colucci, S., & Colasuonno, F. (2007). Fully automated web services discovery and composition through concept covering and concept abduction. *International Journal of Web Services Research (JWSR)*, 4(3), 85–112.

[Reiter, 1991] Reiter, R. (1991). The frame problem in the situation calculus: a simple solution and a completeness result for goal regression. In V. Lifschitz (Ed.), *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy* (pp. 359–380). New York: Academic Press.

[Reiter, 1996] Reiter, R. (1996). Natural actions, concurrency and continuous time in the situation calculus. In L. C. Aiello, J. Doyle, & S. C. Shapiro (Eds.), *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR1996), Cambridge, Massachusetts, USA*, number ISBN 1-55860-421-9 (pp. 2–13).: Morgan Kaufmann Publishers.

[Reiter, 1998] Reiter, R. (1998). Sequential, temporal GOLOG. In A. G. Cohn, L. K. Schubert, & S. C. Shapiro (Eds.), *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR1998), Trento, Italy* (pp. 547–556).: Morgan Kaufmann.

[Reiter, 2001a] Reiter, R. (2001a). *Knowledge in Action, Logical Foundations for Specifying and Implementing Dynamical Systems*. Number ISBN 0-262-18218-1. 292 Main Street, Cambridge, MA 02142, USA: The MIT Press, 1 edition.

[Reiter, 2001b] Reiter, R. (2001b). On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logics*, 2(4), 433–457.

[Reiter & Pinto, 1993] Reiter, R. & Pinto, J. (1993). Temporal reasoning in logic programming: A case for the situation calculus. In *Proceedings of the 10th International Conference on Logic Programming 1993, Budapest, Hungary*, number ISBN 0-262-73105-3 (pp. 203–221).: MIT Press.

[Ross-Talbot & Fletcher, 2006] Ross-Talbot, S. & Fletcher, T. (2006). Web Services Choreography Description Language: Primer. http://www.w3.org/TR/ws-cdl-10-primer/.

[Russell & Norvig, 2002] Russell, S. J. & Norvig, P. (2002). *Artificial Intelligence A Modern Approach*. Number ISBN 0-130-80302-2 in Prentice Hall PTR Series in Artificial Intelligence. Upper Saddle River, NJ 07458, USA: Prentice Hall, Pearson Education, Inc., 2 edition.

[Sardina et al., 2004] Sardina, S., Giacomo, G. D., Lesperance, Y., & Levesque, H. J. (2004). On the semantics of deliberation in IndiGolog from theory to implementation. *Annuals of Mathematics and of Artificial Intelligence*, 41(2-4), 259–299.

[Sardina et al., 2007] Sardina, S., Patrizi, F., & Giacomo, G. D. (2007). Automatic synthesis of a global behavior from multiple distributed behaviors. In J. Collins, P. Faratin, S. Parsons, J. A. Rodriguez-Aguilar, N. M. Sadeh, O. Shehory, & E. Sklar (Eds.), *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI), Vancouver, British Columbia, Canada*, number ISBN 978-1-57735-323-2 (pp. 1063–1069).: AAAI Press.

[Sardina et al., 2008] Sardina, S., Patrizi, F., & Giacomo, G. D. (2008). Behavior composition in the presence of failure. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR 2008), Sydney, Australia* (pp. 640–650).

[Scherl & Levesque, 1993] Scherl, R. B. & Levesque, H. J. (1993). The frame problem and knowledge-producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI1993), Washington, DC, USA* (pp. 689–695). Menlo Park: The MIT Press/AAAI Press.

[Scherl & Levesque, 2003] Scherl, R. B. & Levesque, H. J. (2003). Knowledge, action and the frame problem. *Journal of Artifical Intelligence*, 144(1-2), 1–39.

[Schöning, 2008] Schöning, U. (2008). *Logic for Computer Scientists*. Number ISBN 010-0817647627 in Modern Birkhäuser Classics. Birkhäuser Boston.

[Schubert, 1990] Schubert, L. K. (1990). Monotonic solution of the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In H. E. Kyburg, R. P. Loui, & G. N. Carlson (Eds.), *Knowledge Representation and Defeasible Reasoning*, volume 5 of *Studies in Cognitive Systems* (pp. 23–67). Boston: Kluwer Academic Publishers.

[Shen & Su, 2007a] Shen, Z. & Su, J. (2007a). On automated composition for web services. In *Proceedings of the 16th International Conference on World Wide Web (WWW 2007), Banff, Alberta, Canada* (pp. 1261–1262).

[Shen & Su, 2007b] Shen, Z. & Su, J. (2007b). On completeness of web service compositions. In *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS2007), Salt Lake City, Utah, USA* (pp. 800–807).

[Sirin, 2006] Sirin, E. (2006). *Combining Description Logic Reasoning with (AI) Planning for Composition of Web Services*. PhD thesis, Department of Computer Science, Faculty of the Graduate School of the University of Maryland.

[Sirin et al., 2003a] Sirin, E., Hendler, J. A., & Parsia, B. (2003a). Semi-automatic composition of web services using semantic descriptions. In *Proceedings of the 1st Workshop on Web Services Modeling, Architecture and Infrastructure (WSMAI2003) in conjuction (ICEIS2003), Angers, France* (pp. 17–24).

[Sirin et al., 2004a] Sirin, E., Kuter, U., Nau, D., Parsia, B., & Hendler, J. (2004a). Information gathering during planning for web service composition. In *Proceedings of the Workshop on Planning and Scheduling for Web and Grid Services (ICAPS04), Whistler, Canada*.

[Sirin et al., 2004b] Sirin, E., Parsia, B., & Hendler, J. (2004b). Composition-driven filtering and selection of semantic web services. In *Proceedings of the AAAI Spring Symposium on Semantic Web Services*.

[Sirin et al., 2005a] Sirin, E., Parsia, B., & Hendler, J. (2005a). Template-based composition of semantic web services. In *Proceedings on Agents and the Semantic Web of AAAI Fall Symposium, Virginia, USA*.

[Sirin et al., 2005b] Sirin, E., Parsia, B., & Hendler, J. A. (2005b). Template-based composition of semantic web services. In *AAAI 2005 Fall Symposium on Agents and the Semantic Web* (pp. 85–92).

[Sirin et al., 2004c] Sirin, E., Parsia, B., Wu, D., Hendler, J. A., & Nau, D. S. (2004c). HTN planning for web service composition using SHOP2. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4), 377–396.

[Sirin et al., 2003b] Sirin, E., Wu, D., Parsia, B., Hendler, J., & Nau, D. (2003b). Automatic web services composition using SHOP2. In *Proceedings of Planning for Web Services Workshop (ICAPS 2003) in Trento, Italy*.

[Smith et al., 2004] Smith, M. K., Welty, C., & McGuinness, D. L. (2004). OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/. W3C Recommendation.

[Snabe et al., 2009] Snabe, J. H., Rosenberg, A., Moeller, C., & Scavillo, M. (2009). *Business Process Management the SAP Roadmap*. Number ISBN 978-1-59229-231-8 in SAP Press Series. Galileo Press, 1 edition.

[Sohrabi et al., 2008] Sohrabi, S., Baier, J., & McIlraith, S. A. (2008). HTN planning with quantitative preferences via heuristic search. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Oversubscribed Planning and Scheduling, Sydney, Australia* Sydney, Australia. To appear.

[Sohrabi & McIlraith, 2008] Sohrabi, S. & McIlraith, S. A. (2008). On planning with preferences in HTN. In *Proceedings of the 4th Multidisciplinary Workshop on Advances in Preference Handling (M-PREF 2008)* (pp. 103–109). Chicago, IL, USA: AAAI. A longer version of this paper appeared at NMR08.

[Sohrabi et al., 2006] Sohrabi, S., Prokoshyna, N., & McIlraith, S. A. (2006). Web service composition via generic procedures and customizing user preferences. In *Proceedings of the 5th International Semantic Web Conference (ISWC2006)* (pp. 597–611).

[Steinmetz et al., 2008] Steinmetz, N., Toma, I., Foxvog, D., Keller, U., Kerrigan, M., Kopecky, J., Krummenacher, R., Lausen, H., Sirbu, A., Roman, D., Scicluna, J., Fensel, D., Kifer, M., & de Bruijn, J. (2008). WSML Language Reference. http://www.wsmo.org/TR/d16/d16.1/v1.0/. WSML Final Draft D16.1v1.0.

[Studer et al., 2007] Studer, R., Grimm, S., Hitzler, P., Abecker, A., Preist, C., Lausen, H., Lara, R., Polleres, A., de Bruijn, J., Roman, D., Fischer, S., & Paolucci, M. (2007). *Semantic Web Services, Concepts, Technologies and Applications*. Number ISBN 978-3-540-70893-3 in Semantic Web Services. Tiergartenstrasse 17, D-69121 Heidelberg, Germany: Springer Verlag, 1 edition.

[Thakkar et al., 2002] Thakkar, S., Ambite, J. L., & Knoblock, C. A. (2002). Dynamically composing web services from on-line sources. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI), Workshop on Intelligent Service Integration, Edmonton, Alberta, Canada*.

[Thakkar et al., 2003] Thakkar, S., Ambite, J. L., & Knoblock, C. A. (2003). A view integration approach to dynamic composition of web services. In *Proceedings of the 13th International Conference on Automated Planning & Scheduling (ICAPS 2003), Workshop on Planning for Web Services, Trento, Italy*.

[Thakkar et al., 2004] Thakkar, S., Ambite, J. L., & Knoblock, C. A. (2004). A data integration approach to automatically composing and optimizing web services. In *Proceeding of Workshop on Planning and Scheduling for Web and Grid Services (ICAPS2004)*.

[Toma et al., 2008] Toma, I., Steinmetz, N., Foxvog, D., Keller, U., Kerrigan, M., Kopecky, J., Krummenacher, R., Lausen, H., Sirbu, A., Roman, D., Scicluna, J., Fensel, D., Kifer, M., & de Bruijn, J. (2008). WSML/XML. http://www.wsmo.org/TR/d36/v1.0/. WSML Final Draft D36v1.0.

[van der Aalst et al., 2005] van der Aalst, W., Benatallah, B., Casati, F., & Curbera, F., Eds. (2005). *Business Process Management: Procedings of 3rd International Conference, BPM 2005, Nancy, France*, number ISBN 3540282386 in Lecture Notes in Computer Science LNCS 3649, Springer-Verlag GmbH, Tiergartenstrasse 17, D-69121 Heidelberg, Germany. International Conference of BPM 2005, Springer Verlag.

[van der Aalst et al., 2003] van der Aalst, W., ter Hofstede, A., & Weske, M., Eds. (2003). *Business Process Management: Procedings of 1st International Conference, BPM 2003, Eindhoven, The Netherlands*, number ISBN 3540403183 in Lecture Notes in Computer Science LNCS 2678, Springer-Verlag GmbH, Tiergartenstrasse 17, D-69121 Heidelberg, Germany. International Conference of BPM 2003, Springer Verlag.

[van Emde Boas et al., 1990] van Emde Boas, P., Johnson, D., Seiferas, J., Li, M., Vitanyi, P., Aho, A., Mehlhorn, K., Tsakalidis, A., Schwartz, F. Y. J., Sharir, M., Vitter, J., & Flajolet, P. (1990). *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, volume A. Elsevier Science.

[Weske, 2007] Weske, M. (2007). *Business Process Management, Concepts, Languages, Architectures*. Number ISBN 3540735216 in Business Process Management. Tiergartenstrasse 17, D-69121 Heidelberg, Germany: Springer Verlag, 1 edition.

[Wooldridge & Jennings, 1994] Wooldridge, M. & Jennings, N. R. (1994). Agent theories, architectures and languages: A survey. In *ECAI Workshop on Agent Theories, Architectures and Languages* (pp. 1–39).

[Wooldridge & Jennings, 1995] Wooldridge, M. & Jennings, N. R. (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 115–152.

Index