SPECIFICATION, COMPOSITION AND PROVISION OF TRUSTWORTHY CONTEXT-DEPENDENT SERVICES

NASEEM ISMAIL IBRAHIM

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

JUNE 2012

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Naseem Ismail Ibrahim**
Entitled: **Specification, Composition and Provision of Trustworthy Context-dependent Services**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr. Deborah Dysart-Gale

______________________________ External Examiner
Dr. Abdellatif Obaid

______________________________ Examiner
Dr. Anjali Agarwal

______________________________ Examiner
Dr. Joey Paquet

______________________________ Examiner
Dr. Olga Ormandjieva

______________________________ Supervisor
Dr. Vangalur Alagar

______________________________Co-supervisor
Dr. Mubarak Mohammad

Approved ______________________________
Chair of Department or Graduate Program Director

__________ 20 ______ ______________________________
Dr. Robin A.L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

Specification, Composition and Provision of Trustworthy
Context-dependent Services

Naseem Ismail Ibrahim, Ph.D.
Concordia University, 2012

Ubiquitous computing opened big markets for service provision and consumption bringing benefits for both service providers and consumers. At the same time, it introduced many challenges for developers, such as providing flexible contracts and measurable proofs that assure consumers about the trustworthiness of services. Current approaches for the specification, discovery, and provision of services have not met these challenges. They do not realize the essential relationship between the service contract and the conditions in which the service can guarantee its contract. Moreover, they do not use any formal methods for specifying services, contracts, and compositions. Without a formal basis it is not possible to justify through a rigorous verification the correctness conditions for service compositions and the satisfaction of contractual obligations in service provisions. This thesis makes three major contributions to remedy these drawbacks.

The first contribution is a formal service model, which is called *ConfiguredService*. In this model, service and contract are packaged together. The service part includes functional and nonfunctional aspects of service, and the data parameters and attributes that are essential to define the functional and nonfunctional aspects. The contract part includes business rules, legal aspects, and context information.

The second contribution is *ConfiguredService* composition and verification approach. Several rules for composing *ConfiguredServices* are defined. The verification approach automatically verifies whether or not a stated property is true in service compositions. Since most of the time compositions are required prior to service delivery the verification process enhances trustworthiness at service selection and service provision contexts.

The third contribution is a service provision architecture in which *ConfiguredServices* and their compositions are formally embedded. Service publication, service discovery, service selection and ranking, and service delivery are rigorously defined. The significance here is the way context information is defined for each stage, and is used in the interactions between the different components of the architecture elements in order to sustain the trustworthiness properties at all stages.

To the greatest teachers in my life, ***my parents****.*

ACKNOWLEDGMENTS

I gratefully acknowledge the guidance, invaluable advice, and earnest support of my supervisor and mentor Dr. Vangalur Alagar. I learned from him not only much of what I know about research, but also the importance of being committed, caring and supportive to students. I was truly privileged to be one of his students. I would also like to express my appreciation to my co-supervisor Dr. Mubarak Mohammad for all his advice and guidance. My appreciation is also extended to the members of my supervisory committee.

Last and foremost, I am grateful and will always be indebted to my parents for their unconditional love, affection, patience and encouragements. I would also like to convey my most sincere gratitude to my sisters, who have given me their love and support.

Contents

List of Figures

List of Tables

List of Acronyms

FrSeC	Formal Framework for Providing Context-dependent Services
SRe	Service Registry
SR	Service Requester
SP	Service Provider
PU	Planning Unit
PNU	Plan Negotiation Unit
EU	Execution Unit
TA	Trusted Authority
CGU	Context Gathering Unit
SPL	Service Processing Languages
SRL	Service Registry Language
SQL	Service Query Language
TAL	Trusted Authority Language
SUL	Service Planning Unit Language
NUL	Service Negotiation Unit Language
TADL	Trustworthy Architectural Description Language
CSDL	ConfiguredService Description Language
CSQL	ConfiguredService Query Language
SUDL	Planning Unit Description Language
TADDL	Trusted Authority Description Language
NUDL	Negotiation Unit Description Language
CSST	ConfiguredService Specification Tool

CSPT	ConfiguredService Publication Tool
SQST	Service Query Specification
ET	Execution Tool
PNT	Plan Negotiation Tool
PVT	Provider Verification Tool
RVT	Requester Verification Tool
SCT	Service Composition Tool
CTT	Composition Transformation Tool

Chapter 1

Introduction

Service-Oriented Computing (SOC) [GP08] is a recent computing paradigm that uses *service* as the fundamental element for application development processes. The primary goal of SOC is a rapid, low-cost development of distributed service applications in heterogeneous environments. An architectural model of SOC in which *service* is a first class element is called *Service-Oriented Architecture (SOA)* [Erl07].

In a SOA it should be possible to define, update, compose, communicate, and deliver a multitude of services. Many SOAs for an application are possible, however they may differ in the way they define services, define service operations, and realize them in a practical setting. Yet, all SOAs must agree on the following three distinctive characteristics:

- services are the basic constructional elements,
- service interactions are standard-based, and
- SOA is both dynamic and evolving.

These three characteristics distinguish SOA from traditional software architecture [JYZ$^+$07]. Consequently, SOA demands a new approach to conceptualize services in order to weave them together to meet an application.

Almost in parallel with SOC concept, rapid advancements in computer and electronic engineering have resulted in mass production of hand-held computing devices. In particular, the ubiquitous spectrum of high connectivity has made these devices accessible to a wide range of consumers. In turn, consumers meet their service providers directly in the cyber space with nobody in-between. This has opened a huge heterogeneous customer base for service providers, who provide services through such devices. For example, it is reported in the media that in the span of just 3 years, 500,000 applications offering a wide variety of services to consumers were provided at the Apple store. It is also reported that during this period, 18 Billion downloads happened driving a business of US$ 1.782 Billion in the year 2010. Many other business enterprises, such as Amazon App Store, Google, and Microsoft Market place, also offer pre-packaged services. These technological advancements have brought benefits for both customers and providers of pre-packaged services. As a consequence of cheap technology, it has become possible for small and medium businesses to compete with big corporations, by offering customized services tailored towards a large set of specific groups of customers. This has proved to be advantageous for customers because they can choose better quality services at competitive prices and affordable service guarantees from several service providers. However, customers can only choose from a limited set of simple pre-packaged services.

If the service providers want to expand their customer base and earn economic value, they must offer *rich* services that can be browsed, queried, compared, and composed into complex services. At the same time they must also offer a flexible service contract. Unfortunately, the current service models are awfully inadequate to meet these goals. Towards improving the service model, the service providers should address the following two categories of major challenges:

- *Offering rich service:* How to move away from a simple, pre-packaged service concept to a service that is rich in its content? How to create precise service description that is easy to understand by the customer, and yet is verifiable formally? How to support service descriptions with a formal basis in order that the contract terms are formally analyzable? How to enable the customer to create complex services and safely configure at prescribed contexts?

- *Offering trustworthy service:* How to bundle a service with a set of guarantees (service claims) under a flexible contract? How to provide customer interfaces to validate these claims? How to create services with trustworthiness criteria and embed it faithfully in service publications? How the embedded trustworthiness guarantees can be subjected to a measurable proof of correctness by the customers?

This thesis answers these questions. The answers consist of methods and methodologies, structures and semantics, all illustrated through a case study.

1.1 Research Motivation

SOA has become a de facto software engineering paradigm for developing service-oriented applications. So, it is essential that the research efforts for answering the questions rose earlier to be within the confines of the SOA paradigm. So, we ask the question "*can the current SOA paradigm be used, as is, to develop systems which are trustworthy and context-dependent?* To answer this question, we need to investigate whether or not current SOA approaches can fulfill these requirements.

1.1.1 Status of the Current SOA

A logical view of the current SOA is presented in Figure 1. It represents an abstract view of the interaction process between services. There are three major roles:

- *Service provider*: It is the entity that defines and implements a service. It publishes the descriptions of services (not their implementations) in service registries.

- *Service requester*: It is the entity that browses service descriptions in a service registry and invokes a service. It represents the client side of the interaction. It can be an application or another service.

- *Service registry*: It is a shared medium for enabling the automated publication and discovery of services.

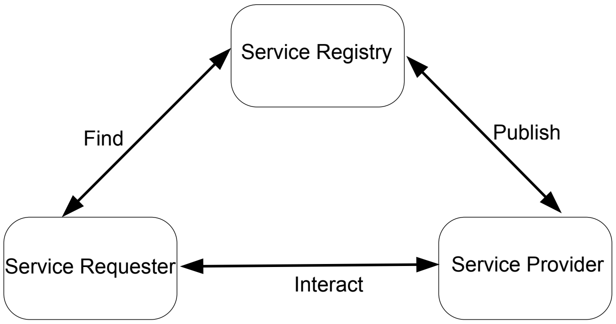


Figure 1: SOA Logical View

The interactions among the roles usually follow three main steps. First, service providers publish their service descriptions through service registry server. Second, the service requester can search the service registry server looking for a suitable service. Third, the service requester can directly interact with the service provider by sending messages to it. This interaction is regulated by a *contract* [TP05], which is a formal agreement between collaborating entities and supporting parties.

According to [Erl07], a service contract establishes the terms of engagement with the service, provides technical constraints and requirements, and any semantic information the service provider wishes to make public. The essential structural aspects of a service contract [OR08] are *interface*, *functionality*, *protocol*, and *quality of service* (nonfunctional) requirements.

- *Interface*: It defines the syntactic communication abstraction for service request and service response.

- *Functionality*: It precisely states what a service can do for a user. It is the set of operations provided by a service. Each operation can be specified using *preconditions* and *postconditions*. All *preconditions* must be true when a service operation is called. All *postconditions* must be guaranteed to be true after the service is successfully invoked.

- *Protocol*: It is the behavior of the service in terms of the input messages (requests) and the output messages (responses).

- *Nonfunctional properties*: Quality of service features are the nonfunctional properties of the service. In principle, these include *performance*, *reliability*, *availability*, *security*, and *accessibility*.

Service composition models are central to SOA. Currently, a composite service is regarded as coordinated aggregate of services [Erl07]. Both static and dynamic compositions are allowed. While static composition is done at design-time, dynamic composition is performed at service execution time. Its goals are to (1) satisfy user contexts (e.g., location, time, and profile) and user preferences [FS09], (2) change the functionality by adding or removing services at run time [EAS08], and (3) find other compatible services or change the process definition and redesign the system to deal with user issues [DS05].

1.1.2 A Critique of the Current SOA Models

In SOA *context* plays an important role, right from the publication of service to its final delivery. Context was not considered as part of SOA by many researches. All those who included context in SOA did so only informally, mainly using context information as a filter during service discovery. It was not used ever to constrain the contract and service delivery.

Some attempt has been made in the past to include trustworthiness properties, and especially in the semantic web domain, for services. But no systematic method exists to represent trustworthiness properties as part of service description. Some of the current SOA models are augmented with informal descriptions of a partial set of trustworthiness properties. As such, no formal analysis can be done to validate the service quality of a published service.

The current structure of service contract is inadequate in dealing with changing contextual service execution and service delivery situations. If context information changes in between service discovery and service delivery, then such changes have the potential to invalidate many business rules governing a business service. In such situations the contract, unless renegotiated dynamically, becomes null and void. Moreover, trust itself should be split into two parts, one part governing the service provider and the other part governing the service claims of the provider. The former is static, whereas the later is dynamic.

Current service modeling approaches focus mainly on the functional part of the service. As a consequence, the current composition methods are just functional compositions. No theory for composition exists. As a result, in the current SOA it is not possible to define the contract that binds a composed service, let alone verify that it satisfies the composite service.

1.1.3 Recent Developments in Formal Modeling of Context and Trustworthiness

From the critique section follows the key decision to include context and trust in modeling SOA. Also one of the goals is to be able to formally verify service properties in different contexts. Thus, both context and trustworthiness should be formally representable in a SOA model. In this section we motivate our choice of notation for modeling context and trustworthiness.

Context

In SOA, the provision of services is strongly related to the contextual information. The contextual information includes the context of the service requester and the context of the service provider. The service provider can guarantee its contract in specific situations. These situations will act as conditions that should be true for the service provider to guarantee its contract. In order to enable formal analysis of such contextual constraints, we need to formalize the relationship between service contracts and the service provision context.

Context has been defined [Dey01] as the information used to characterize the situation of an entity. This entity can be a person, a place, or an object. This entity is relevant to the interaction between a user and an application. A system is *context-aware* if the context information is used by it to better provide information or services to its users [Dey01]. The context representation proposed by Wan [Wan06], and the logic of context proposed in [WA08a] for reasoning about context-awareness are suitable formalisms for enriching

SOA modeling.

Trustworthiness

In published service contracts, we believe that trust should be stated and should be divided into two parts. The first part is concerned with the claims made on the service itself. The second part is related to the service provider providing the service. Trustworthiness properties related to the service should include safety, security, reliability and availability. Trustworthiness properties related to the service provider should include recommendations from independent organizations, or consumers' ratings and reviews.

In the literature, trustworthiness is defined as the system property that denotes the degree of user confidence that the system will behave as expected [SBI99] [ALRL04]. The terms trustworthiness and dependability are used interchangeably [Som07]. Dependability is defined as "the ability to deliver services that can justifiably be trusted" [ALRL04]. A comparison between the two terms presented in [ALRL04] has concluded that the two properties are equivalent in their goals and address similar concerns. The goals of dependability are providing justifiably trusted services and avoiding outage of service that is unacceptable to the consumer.

There is a common consensus [SBI99] [ALRL04] [MdVHC02] that trustworthiness is best expressed as a composite concept of safety, security, reliability, and availability. These properties have been formalized [Moh09] for developing trustworthy systems. We can adapt them to services, as explained below.

- **Safety** is the quality of the operational behavior of the system in which no system action (service) that may lead to catastrophic consequences will be triggered. Safety includes timeliness properties that describe time constrained service execution behavior.

- **Security** is a composite property that includes confidentiality and integrity. Confidentiality ensures that system services and information (data) are not disclosed to unauthorized users. Integrity ensures that there is no improper alteration to the system state or the published service information.

- **Reliability** is the quality of continuing to provide correct services despite any failure. It is possible to have an accepted frequency of failures. Reliability is defined as the guaranteed maximum number of failures in a unit of time.

- **Availability** means readiness to provide correct service in a user specified context. It is the quality of operation in which there is no unforeseen or unannounced disruption of service. A temporary outage of service may not cause big problems for a non-critical system. The required services can be requested at a later point of time when the system becomes available. However, any service outage for a safety-critical system may lead to catastrophic consequences. When a system fails, availability specifies the maximum accepted time of repair until the service returns back to operate correctly.

1.1.4 Research Directions - Summary

In Section 1.1.2 we made it clear that currently there is no approach for the specification, verification, publication, discovery, selection, and composition of services that takes into consideration the relationship between the service contract and the related contextual information. However, the relationship between the service contract and the related contextual information on which the service can guarantee its contract plays a crucial role in service provision. Thus, a new service modeling and a new composition theory are required. Formal methods for the specification of the services, their contracts, the related contextual information, and their composition are essential in order to conduct formal verification of service claims. This has led us to borrow formal notations of context and trustworthiness in meeting this objective. These two decisions lead us to the following three major research directions.

- *Enrich Service Modeling:* Service functionality is to be bundled with its trustworthiness features, contract that is flexible, and context information of service provider and service requester. This enriched service will have to be formalized, and analyzed at different stages of service processing.

- *Develop a Composition Theory:* Composition theories are essential, in order to validate composite service functionalities against their new contracts. A formal semantic basis for composition is necessary. Methods for both static and dynamic compositions should be provided.

- *Service Processing Framework:* Rich services are intended to be analyzed, certified, published, composed, discovered and delivered. These operations should enable the end-to-end processing of services in a trustworthy manner.

1.2 Research Contributions and Thesis Outline

In following the three research directions defined above, we arrived at the formal framework *FrSeC*, the *formal framework for the provision of context dependent services*. This framework comprehensively supports all the intended SOA activities that we desired. It supports specification, verification, publication, discovery, selection and composition of rich services with context-dependent contracts. The research methodology discussed in Chapter 3, identifies specific research steps along each research direction, raises questions with regard to the research issues in each step, and answers them in some depth with the specific methodology to be followed in producing a solution to each question. The major contributions of the thesis arise from their investigation, and are organized as follows.

- *Related Work:* Chapter 2 presents a comparative study between our work and related published work in SOA, and brings out the relative merit of our work.

- *New Service Model:* A formal model for the specification of services with context-dependent contracts is given in Chapter 4. This chapter includes a discussion on the semantic analysis of services, and flexible contracts.

- *Static Service Composition:* Chapter 5 introduces a static composition theory for our service model. It is intended to be used by service providers. Compositions will take into account the entire service description, namely functional and nonfunctional properties, trustworthiness properties, legal rules and context information.

- *Formal Verification:* This subject is spread through two chapters. In Chapter 4 we discuss the types of verification necessary to analyze service models. Chapter 6 presents a novel approach for the formal verification of essential contract properties in service compositions. This approach is automated, and currently is enabled by the UPPAAL model checker.

- *Formal Framework:* Chapter 7 introduces the *FrSeC* and its components, formally describes the components, and their roles. Query types and service discovery with respect to query types are discussed at length.

- *Dynamic Service Composition:* Chapter 8 discusses dynamic service compositions in response to composite service queries supported by *FrSeC*. Corresponding to each query composition type there exist exactly one service (static) composition method.

- *Development Methodology:* Chapter 9 discusses a development methodology for trustworthy context-dependent service-oriented applications using *FrSeC*. It introduces the languages required in the development process.

- *Conclusion and Future Work:* The thesis concludes in Chapter 10 with a summary, an assessment of the presented approach, and a list of ongoing research projects related to this thesis.

Chapter 2

A Brief Survey of Related Work

Research in SOC has produced a large volume of work ranging from pure business perspectives to pure software engineering perspectives. We narrow down this spectrum of work for a comparison with our work, by filtering it in the four dimensions *service modeling*, *formalism*, *composition methods*, and *service provision framework design*. Along each dimension the discussed approaches are compared with respect to the specific manner they (1) handle the modeling of *service functionality*, *nonfunctional properties and trustworthiness*, *legal rules*, *context*, (2) use *formalism* and *formal Verification*, and (3) provide *tool support*.

2.1 Service Modeling

This section briefly sketches the state of the art in service modeling. The modeling approaches can be classified based either on the language, the architecture or a combination of both used to describe service. The two main languages that have been used for modeling services are UML [MSK08, soa08], and WSDL with the related Web description languages [WSD, MPM$^+$04, ZkMM06, RKL$^+$05]. Architecture based service modeling approach uses an Architectural Definition Language (ADL) [JYZ$^+$07, DST$^+$06] to describe services. There are a few methods [FLB06, CMX08] which use both language and some abstract architectural details for describing service features.

2.1.1 Languages for Service Modeling

The two major languages that have been used for service modeling are UML and Web services family.

UML Extensions

UML4SOA UML is a general purpose software modeling language and is not meant for modeling services. By introducing several extensions to UML, the language UML4SOA was created in 2008 as part of the SENSORIA [WBF$^+$08] project. It utilizes the extension mechanisms provided by UML2. It follows a minimal extension principle. That is, existing UML constructs are used wherever possible and new model elements are defined to model service-oriented features only when necessary. UML4SOA provides model elements for structural and behavioral aspects, business goals, policies and nonfunctional properties of SOAs. The *Model-Driven Development for Service-Oriented Architectures* (MDD4SOA) [MSK08] includes UML4SOA for modeling services and a few model transformation tools for the generation of code in various output languages. The transformation process is performed in two steps:

- The UML4SOA model is transformed to an intermediate model called, IOM (Intermediate Orchestration Model). The control flow of the UML4SOA is analyzed during this step.
- The IOM model is transformed to a PSM (Platform Specific Model). The PSM models can be one of the following: (1) Web services standards such as BPEL and WSDL, (2) the object-oriented language Java, (3) the formal language JOLIE [MGLZ07].

There exists no precise guideline for creating extensions for UML2. Consequently, there is no standard semantics for extended notations. These approaches lack formalism and focus mainly on using diagrams.

SoaML The Service oriented architecture Modeling Language (SoaML) [soa08] is another extension of UML2, developed by the Object Management Group (OMG). It describes a UML profile and meta-model for the design of services within SOA. The main goal of SoaML is to support the activities of service modeling and design, and to fit into an overall model-driven development approach (MDA). It separates the logical implementation of a service from it possible physical realization on various platforms. This separation helps simplifies the service model and make it more flexible. The service modeling features include (1) specification of systems of services, (2) specification of individual service interfaces, and (3) specification of service implementation.

Web Services Family of Languages

WSDL The de facto language for describing Web services is *Web Services Description Language* (WSDL) [WSD]. It is an XML-based language for specifying the data and operations that represent a Web service contract. WSDL is supported by a wide range of tools. However, the meaning (semantics) of data cannot be specified in WSDL. This lack of semantics, makes it necessary for humans to be involved for automated service discovery and composition within open systems [KBM08]. Semantic Web services (SWS) [KBM08] remedies this situation by enabling the semantic embedding for data in WSDL.

Thus, with SWS the functionality of a Web service is bound to its semantic annotations [NMFR09] which enables Web services to be automated. SWS includes the languages OWL-S [MPM$^+$04, KBM08] and WSMO [ZkMM06].

OWL-S OWL-S [MPM$^+$04] [KBM08] is the Web Ontology Language. It is structured into three sub-ontologies, for defining different semantic aspects of Web services. The first aspect is the Web service functionality, including the constraints and nonfunctional properties that influence it. This is defined using the ServiceProfile. The second aspect is ServiceModel which tells a service requester how to use the service, by detailing the semantic content of requests, the conditions under which particular outcomes will occur, and the step by step processes leading to those outcomes. The third aspect is the ServiceGrounding which maps elements in the ServiceModel to their corresponding WSDL description.

WSMO Web Services Modeling Ontology (WSMO) is a full-fledged framework for Semantic Web services (SWS). It aims to enhance the syntactic description of Web services with semantic metadata [ZkMM06]. The three components of WSMO are (1) a formal specification component of concepts for SWS, (2) Web Services Modeling Language (WSML) in which WSMO concepts are defined, and (3) Web Services Execution Environment (WSMX) which executes SWS. WSMO requires *Ontologies*, *Goals*, *Web Services* and *Mediators* [RKL+05] to be defined in order to define Semantic Web services. *Ontologies* [ZkMM06] provide a way to help in querying for knowledge of the Web by associating information with descriptions of its meaning. Ontologies define descriptions of the things that exist in a domain of interest with the relationships that exists between those things. The Web Service element of WSMO provides a conceptual model of all the aspects of a Web service, including its functionality, nonfunctional properties, and interfaces. A well defined model of Web services with well defined semantics can be processed by computers without human intervention. Capability defines the functional aspects of the actual service. Interface defines the behavioral aspects of the service. It contains, *Choreography*, which defines the interface for consumption and the *Orchestration* which defines how functionally can be achieved. *Goals* in WSMO are used to describe the desires of users. A service requester uses Goals section to represent the service they want by specifying its capability. The difference between Web Service descriptions and Goals is that Web Service descriptions are intended to provide descriptions of the mechanics of how a service provides its capability and behavior, while Goal descriptions describe what capability and behavior the requester would like to get. Goals are described in terms of ontologies used by the requesters. *Mediators* are responsible of handling heterogeneity between Goals and Web Services, by resolving possible mismatches that occur between resources that should be interoperable.

2.1.2 Architecture Description Languages for Service Modeling

We review three languages in this section. *SOADL* [JYZ$^+$07] [DST$^+$06] is an architecture description languages (ADL) which uses Pi-calculus formalism for service modeling, composition, and verification. The other two languages are SRLM [FLB06] and SOFM [CMX08]. These two languages are hybrid in the sense that they use information outside an architecture to model services.

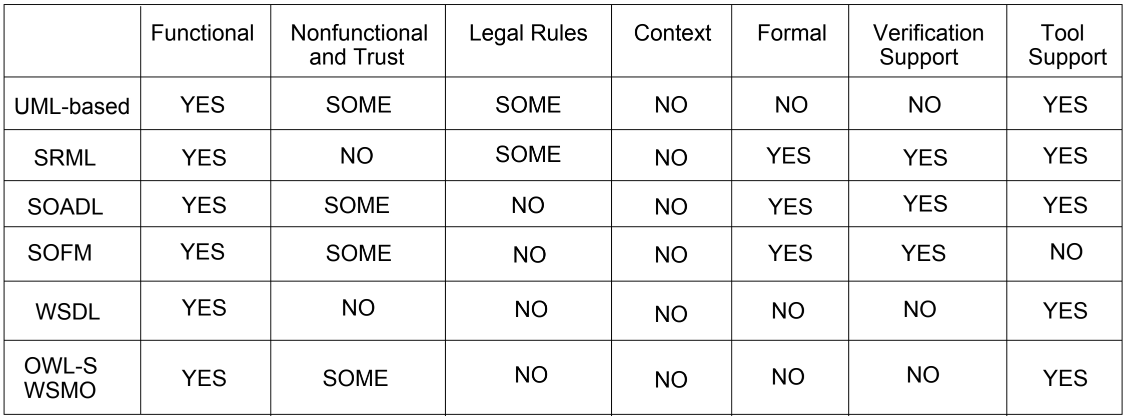
SOADL The service oriented description language (SOADL) is an example of ADL based approach. It specifics the interfaces, behavior, semantics, and quality properties of services. It models and analyzes the dynamic and evolving architecture. It can be used to design service oriented system and to define service composition in a simple way. It uses XML as a meta-language. OADL defines the architecture in a hierarchical way. Services are of two kinds: *atomic* and *composite*. An *atomic* service performs a basic business function. A *composite* service is composed from a set of atomic or composite services. SOADL uses Pi-calculus to model the behavior of the services formally. It uses Pi-calculus tools to verify the correctness of the composition results. It also defines transformation rules that can manually transform the system to BPEL representation.

SRML Service Component Architecture (SCA) [MR09] [Cha07] describes a model for constructing applications and systems using SOA. It extends on prior approaches to implement services, and builds on open standards, such as Web services. Many modeling approaches have been inspired by SCA. The SENSORIA Reference Modeling Language (SRML) [FLB06] is one of them. It provides semantic modeling primitives for service-oriented systems that are independent of the languages and platforms in which services are programmed and executed. It uses some formalism to model the computation and coordination of services. The main novelty of SRML is that it adopts a set of complex primitives tailored specifically for modeling the business conversations that occur in service-oriented computing. SRLM services are characterized by (1) the conversations, and (2) the properties of those conversations [dA09]. In SRML, the unit of design is a *module*. The two types of modules are *activity module* and *service module*. Activity modules define applications that are developed to satisfy a specific requirement of a business organization, and not to be published as a service. On the other hand, a *service module* is developed for publication. A module is defined in terms of a number of entities and how they are interconnected. *Wires* interconnects those entities. Each *wire* defines an interaction protocol between two entities. SRML defines internal and external configuration polices. The internal policies define the initialization and termination conditions of each component and the conditions that trigger the discovery process of each external service. The external polices express constraints for Service Level Agreements (SLA).

SOFM The Service-Oriented Feature Model (SOFM) [CMX08] captures the *service features* provided by an application in an abstract form. A *service feature* represents the requirements of an application as a collection of services. The two main concepts of SOFM are service features and their relationships. The feature model for an application is created hierarchically, by classifying and structuring service features. Service features can be classified in two categories: *constraints* and *refinements*. Constraint category is the set of static relationships between service features. It is further classified according to three types of relationships. If selecting a service feature requires the selection of another service feature then they have *Require* relationship. If selecting a service feature means excluding another feature then they have the *Exclude* relationship. The relationship *Other constraint* indicates rationales, composition rules and trade-off that developers made when developing the application. *Refinements* is a binary relationship between two service features indicating that the first service feature implies the second one. The three kinds of refinements are (1) *Decomposition*, which arises while refining a service feature to its constituent service features, (2) *Specialization* which is a refinement of a service feature with further details, and (3) *Operationalization* which refines a service feature into its solution in the target system.

2.1.3 Analysis

The discussed approaches are compared and the result of comparison is presented in Figure 2. It is clear that all approaches support the modeling of the functional behavior. Nonfunctional and trustworthiness properties are only supported in a simple manner by few approaches. Contextual information is not represented by any approach, hence the relationship between contract and context is totally ignored. A couple of approaches, which have ignored the modeling of nonfunctional and trustworthiness properties, have used formal methods and conducted formal verification. Except for UML and Web services languages most of the approaches provide a minimum amount of tools to support the modeling using their service models.



	Functional	Nonfunctional and Trust	Legal Rules	Context	Formal	Verification Support	Tool Support
UML-based	YES	SOME	SOME	NO	NO	NO	YES
SRML	YES	NO	SOME	NO	YES	YES	YES
SOADL	YES	SOME	NO	NO	YES	YES	YES
SOFM	YES	SOME	NO	NO	YES	YES	NO
WSDL	YES	NO	NO	NO	NO	NO	YES
OWL-S WSMO	YES	SOME	NO	NO	NO	NO	YES

Figure 2: Service Models Comparison

2.2 Service Composition Approaches

Service composition is understood in SOC to be a process that aggregates services to create new services that provide complex functionality. In this section, we have chosen to discuss two types of service composition approaches. These are (1) Web services based approaches, and (2) formal methods-based approaches.

2.2.1 Web Services Approaches

The two main approaches for syntactic service composition are *orchestration* and *choreography* [tBBG07b]. In *Orchestration* approach an orchestrator is responsible for invoking and combining the activities for a composition. The *choreography* approach composes services by defining conversations that should be undertaken by each participant service. The overall activity is achieved by the composition of interaction among the collaborating services. BPEL is the most important orchestration approach while WS-CDL is an example of choreography approach. The main difference between BPEL and WS-CDL is that WS-CDL describes a global view of the observable behavior of message exchanges of the participating service, while BPEL describes the behavior from the view point of the orchestrator [tBBG07b].

BPEL

Business Process Execution Language (BPEL) [CKM$^+$03] is an XML based language that was designed to enable the coordination and composition of services. It has emerged as the standard to define and manage composition for Web services.

It is based on Web services Description Language WSDL. BPEL uses a workflow-based approach to provide behavioral extension to WSDL. BPEL defines relationships by invocations using control and data flow links. *Process* is the main construct to model the flow of services. It is a concurrent description that connects *activities* that send and receive messages. External Web services providers are defined as *port* of a particular *port type*. A port type has a WSDL description. *Partner links* are used to specify which activity is linked to which port provider [tBBG07b].

The basic element in the BPEL process is called an *activity*, it can either be a primitive or a structured activity. *Primitive* activities contains 1) *invoke*, which invokes an operation of some Web service, 2) *receive*, which waits for a message from an external source, 3) *reply*, which reply to an external source, 4) *wait*, which wait for some time, 5) *assign*, which copy data from one place to another, 6) *throw*, which indicate errors in the execution, 7) *terminate*, which terminate the entire service instance, and 8) *empty*, which does nothing.

Structured activities contains 1) *sequence*, which defines an execution order, 2) *switch*, which is used for conditional routing, 3) *while*, which is used for looping, 4) *pick*, which is used for race conditions based on timing or external triggers, 5) *flow*, which is used for parallel routing, and 6) *scope*, which is used for grouping activities.

WS-CDL

The Web services Choreography Description Language (WS-CDL) [WC] is an XML based language that describes peer-to-peer collaborations of participants. It defines the participant's common and complementary observable behavior where ordered message exchanges result in accomplishing a business goal. *Interaction* activity is the most important construct of WS-CDL. It describes an information exchange between parties, with a focus on the receiver. Interaction consist of *participants*, *information* and *channel*. *Participants* correspond to members involved in the choreography. *Information* corresponds to the data being exchanged between participants. *Channels* corresponds to the pipe through which information is exchanged. WS-CDL also supports exception handling and compensations through *exception* and *finalizer*.

2.2.2 Approaches Based on Formal Methods

One of the challenges of SOA is in guaranteeing the correct interaction of services. Because of the message-passing nature of service interaction many errors might occur when services are composed, such as messages not being received, deadlocks and incompatible behaviors. These problems are not new in distributed applications. However, they gain an extra importance in SOA because services can be used by other services rather than just by humans, and SOA encourages the automatic interaction between services [tBBG07b].

Formal methods have the advantage of the support of tools to verify the correctness of service compositions. Formal methods and their tools can be used to check if services are equivalent. It can also be used to check if services and their composition satisfy certain properties [tBBG07b].

Many approaches have used formal methods to model service oriented systems. Those approaches can be categorized depending on the formal approach they follow. Below we review the most important approaches that are respectively based on the formalisms *Automata*, *Petri nets*, and *Process Algebras*. We discuss only how compositions are done, assuming the formal notations of the respective formalisms.

Automata

Many authors [MKB07], [FUMK03], [KPP06], [FBS04], [DCP$^+$06], and [DLSZ06] have used automata to model services and/or their compositions. We briefly explain the two kinds of approaches, selecting three works from this list.

In [FBS04], the authors present an approach to analyze and verify the functional behavior of composite Web services defined using BPEL. The verification is done in two steps. First, the BPEL specification is transformed to an automaton. In the second step the automaton is translated to Promela, the language supported by the model checker SPIN [BA08], which will then be used for verifying the properties. A similar approach is also used by [FFK05] and [YPCG05].

In [DCP$^+$06], the authors show how service composition written in WS-CDL can be automatically translated to a timed automat that can be verified by the model checker tool UPPAAL [BDL04b]. The compositions in WS-CDL can only specify the functional behavior of the participating services. Hence, UPPAAL is only used to verify the correctness of the functional behavior of the composition. UPPAAL is also used in [DLSZ06], to automatically verify systems modeled in the orchestration language Orc [Orc]. The authors define formal timed-automata semantics for Orc expressions.

Petri nets

From the many published studies we have chosen two categories of work to review. One approach is to transform language models to Petri nets, and the second approach is to enhance Petri nets directly for service compositions.

From Language Models to Petri Nets: In [OVvdA$^+$07], the authors show how to map BPEL specification into Petri nets. They automate this transformation by a tool called BPEL2PNML. The output of this tool can be verified and analyzed using the tool Wof-BPEL. In [NM02], the authors define the semantics for a relevant subset of DAML-S (now OWL-S) in terms of a first order logic language. Using this semantics they encode service models into Petri Net formalism. They provide a tool to describe this transformation and verify service composition. In [HSS05], the authors present a Petri net semantics for BPEL. The semantics cover the standard and exceptional behavior of BPEL. A tool that translates the BPEL specification into the input language of the Petri net model checking tool LoLa [Sch00] has been developed. In [RBHJ06], the authors presents a Petri net framework for Web services orchestrations, including both functional and QoS aspects. This is done by translating Orc specification into colored Petri nets. A tool has also been developed to support this approach. This tool takes as inputs the Orc description and the QoS distributions of the sites involved in the orchestration. Monte-Carlo simulation is performed to study the orchestration's QoS dimensioning.

Direct use of Petri Nets: In [HB03], the authors define a Petri net-based algebra for Web services composition. The formal semantics of the composition operators is described in terms of Petri nets. They present how to use their approach to perform performance analysis. In [YK04], the authors present a Petri-net-based unified specification model for conversation protocol and composition. They propose a method for the composition of Web services. They assure the correctness of composition by formal verification. In [ZCCK04], the authors present WS-Net which is an executable architectural description language incorporating the semantics of Colored Petri-net (a generalization of Petri nets that can deal with recursion and data handling) with the style and understandability of object-oriented concepts. It aims to facilitate the verification and monitoring of Web services integration.

Process Algebras

In *Process algebras*, process behavior of concurrent systems can be specified and verified. This formalism has a rich theory on bisimulation analysis which helps to check (1) the ability of one service to substitute another in a composition, and (2) for the redundancy of services. The π*-calculus* [MPW92] is a process algebra that has been found attractive to modeling Web services. The main reasons for that are [tBBG07a]:

- It is formal which provides the ability for the automatic verification of the behavior properties.
- It provides constructs to compose activities in terms of sequential, parallel, and conditional execution.

Below are some examples of process-algebraic approaches to specify and verify service composition.

COWS Calculus for Orchestration of Web Services (COWS) [Tie09] is a formal language for specifying and combining services while modeling their dynamic behavior. COWS design is influenced by process calculi and BPEL. COWS contains features that are borrowed from process calculi, such as not binding receive activities, asynchronous communication, polyadic synchronization, pattern matching, protection and delimited killing activities.

COWS aims to be technology agnostic and not tightly coupled to any Web services technology. It has been defined as a formal language to work at a higher level than BPEL for specifying the business process. This explains the high dependency between BPEL and COWS. It focuses on defining the behavioral aspects of services.

A number of tools have been developed to analyze COWS specifications. An example is a logic and model checking tool that has been developed to check the functional properties of services.

SCC *The Service Centered Calculus (SCC)* [BBN$^+$06] is a general purpose calculus for services which focuses on sessions. It is a name-passing process calculus in which services can be created and invoked. A new session is produced when a service is invoked. This session models the interaction between the clients and the services. It presents a basic mechanism for service orchestration, which has been inspired by Orc's pipeline construct. SCC combines the service oriented flavor of Orc with the name passing communication mechanism of π-calculus.

SOCK *The Service Oriented Computing Kernel (SOCK)* [GLG$^+$06] is a three-layered calculus which addresses all the basic mechanisms for service interaction and composition. It divides design issues into three fundamental parts:

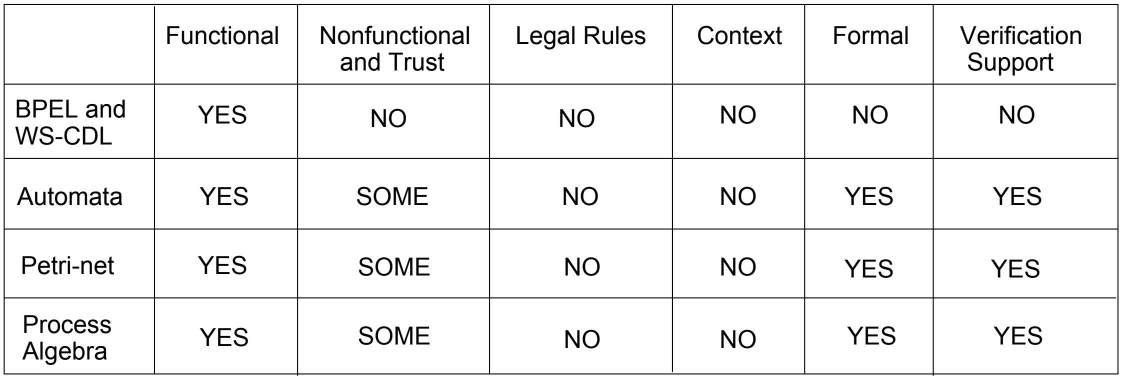
- *Behavior*, which represents the workflow of a service instance,
- *Declaration*, which introduces the aspects related to the execution modalities, and
- *Composition*, which allows to reason about the behavior of the system composed.

SOCK uses correlation information to compose services. This correlation information allows a flexible mechanism to manage relationships among interacting partners.

cc-pi *The concurrent constraint pi-calculus (cc-pi)* [BM08] is a constraint-based model of QoS negotiations for concluding Service Level Agreements. It combines basic operations of concurrent constraints programming with a symmetric, synchronous mechanism of interaction between senders and receivers.

2.2.3 Analysis

The discussed approaches are compared and the result of this comparison is presented in Figure 3. It is clear that all approaches support the composition of the functional behavior. Figure 3 states "SOME" under "Nonfunctional and Trust" because the composition of nonfunctional and trustworthiness properties is only supported in a simple manner by some examples of each formalism. Examples of such approaches are [RBHJ06] for Petri-nets and [BM08] for Process algebra. Contextual information is not considered by any approach. With the exception of Web services approaches, all investigated approaches are formally based and hence supports the formal verification of the composition result.



	Functional	Nonfunctional and Trust	Legal Rules	Context	Formal	Verification Support
BPEL and WS-CDL	YES	NO	NO	NO	NO	NO
Automata	YES	SOME	NO	NO	YES	YES
Petri-net	YES	SOME	NO	NO	YES	YES
Process Algebra	YES	SOME	NO	NO	YES	YES

Figure 3: Comparison of Service Composition Approaches

2.3 Service Provision Frameworks

This section presents a review of some of the service provision frameworks selected from the literature. We don't claim that this list is comprehensive but we believe it is representative of the approaches available in the literature.

2.3.1 SeGSeC

In [FS09], the authors presents a semantics-based context-aware dynamic service composition framework. The framework aims to allow users to request applications in a natural language. The framework models the semantics of services and composes applications based on the semantics of the services. The framework consists of *Component Service Model with Semantics (CoSMoS)*, *Component Runtime Environment (CoRE)*, and *Semantic Graph based Service Composition (SeGSeC)*.

CoSMoS

The Component Service Model with Semantics (CoSMoS) is an abstract component model. It is designed to model the functions, semantics and contexts of components. It also models contexts of users and user-specified rules. CoSMoS models the information as a semantic graph, that is, a directed graph that consists of labeled nodes and links. Since CoSMoS is an abstract model, it can be described in different formats, such as, WSDL and RDF.

CoRE

Component Runtime Environment (CoRE) is a middleware to facilitate the semantics-based context-aware dynamic service composition on various distributed computing technologies. It aims to implement functionalities to (1) discover and execute distributed components, (2) create and manage user components which represent actual users, and (3) acquire contexts of components and users and model them in CoSMoS.

SeGSeC

Semantic Graph-based Service Composition (SeGSeC) is a service composition mechanism. It composes an application requested by a user by synthesizing its workflow. It allows users to request applications using a natural language sentence using the semantic information of components.

SeGSeC adapts to different users and to dynamic environments by using contextual information of components and users. To adapt to different users, SeGSeC supports two kinds of context-aware service compositions:

- *Rule based*: SeGSeC allows a user to specify rules on which to use or not to use a component in a specific context using a natural language. For example, *"If I am in a meeting, do not use a computer"* is a context rule. It applies the user-specified rules when synthesizing workflows.

- *Learning based*: SeGSeC proactively learns user preferences from history information and applies the result of the learning when synthesizing workflows.

SeGSeC implements seamless service migration. It dynamically creates a new workflow upon detecting context changes or upon detecting a failure in the execution of some components in the workflow, and changes the execution status from the old workflow to the new one.

2.3.2 eFlow

eFlow [CIJ$^+$00] is a platform developed by HP Laboratories, for the specification, enactment and management of composite services. It is template based process model that defines the composite service as a process schema that describes the notion of a generic service node which includes a runtime configurable parameter [EAS08]. The composition is defined by a graph that defines the order of execution of the nodes in the process. This graph includes:

- *Service nodes*: They represent the invocation of an atomic or composite service.
- *Event nodes*: They enable service processes to send and receive several types of events.
- *Decision nodes*: They specify the alternatives and rules controlling the execution flow.
- *Arcs*: They denote the execution dependency among the nodes.

The graphs should be specified manually. *eFlow* automatically binds the nodes with concrete services. A service node contains a search recipe that can be used to search for actual services at instantiation time or at run time. When the service node is started, the search recipe is executed, and then a reference to a specific service is returned.

2.3.3 SELF-SERV

SELF-SERV [SBDM02] is a platform for rapid composition of Web services. Web services are declaratively composed and the resulting composing services are executed in a peer-to-peer and dynamic environment. Composite services are defined using state-charts, data conversion rules, and provider selection policies that are translated into XML document for interoperability. The significance of SELF-SERV is the peer-to-peer service execution model. The coordination responsibility is distributed across several peer software components called *coordinators*.

2.3.4 SHOP2

SHOP2 [WPS$^+$03] [VR04] is a domain independent artificial intelligent (AI) planner. SHOP2 uses a hierarchical task network (HTN) to decompose an abstract task into a group of operators that form a plan to implement the task. Planning progresses as a recursive operation, decomposing tasks into subtasks until the primitive tasks that can be performed directly are reached. In the case where the plan later turns out to be infeasible, SHOP2 will backtrack and try other applicable methods.

The composition process of Web services is encoded as a SHOP2 planning problem. SHOP2 is applied for automatic composition of Web services which are provided with DAML-S [BHL$^+$02] (currently OWL-S) descriptions. Web services are described in DAML-S as a process in terms of inputs, outputs, preconditions and effect. A translation is then made from DAML-S to SHOP2 where the composition problem will be dealt with as an AI planning problem.

2.3.5 SWORD

SWORD [PF02] is a developer toolkit for building composite Web services using rule-based plan generation, developed at Stanford University. SWORD does not use service description standards such as WSDL and OWL-S. It uses Entity-Relation (ER) models to specify Web services.

A service is modeled by its preconditions and postconditions. In order to create a service composition, the requester specifies the initial and final states. The rule based expert system then generates the plan following AI planning techniques. The expert system is designed for automatic static composition, and it can be used for dynamic scenarios.

2.3.6 Argos

Argos [AW05] [AGG$^+$05] is an approach to automatically generate computational workflows for service composition problems. Argos relies on an ontology of the application domain to provide formal semantics to the sources and operations available. This ontology is expressed in RDF/RDFS and is used to express the inputs and outputs of the services.

Argos describes the ontology, the sources, and operations as a Triple logic program. The Triple logic engine is used to formally represent the ontology and the services, and to automatically generate the workflows. The workflow is then translated to BPEL for execution.

2.3.7 Composer

In [SSP04], the authors present a semi-automatic method for Web service composition. The system they propose has two basic components a *composer* and an *inference engine*. The *inference engine* stores the services information in its Knowledge Base (KB) and it is used to find matching services. The inference engine is an OWL reasoner built on Prolog. Ontological information is written in OWL and is converted to RDF triples and loaded to the KB. The engine has built-in axioms for OWL inferencing rules. These axioms are applied to the facts in the KB to find all relevant entailments. The *composer* is the user interface that is responsible for the communication with the user.

When a user requests a Web service, the system presents to the user all possible services that match the required service. Services are selected based on the functional and nonfunctional properties. Those properties are presented by OWL classes and the inference engine is applied to match the services requested. In a composition, the match between services is defined between two services that an output parameter of one service is the same OWL class or subclass of an input parameter of another service. If the system found more than one service as a match, it filters the services based on the nonfunctional properties that are specified by the user as constraints. The services that meet the nonfunctional requirements are presented to the service requester.

2.3.8 FUSION

FUSION [VDD$^+$03] is a framework for dynamic Web service composition and automatic execution. FUSION takes a user specification and it automatically generates a correct and optimized plan. It then executes this plan to verify the results and make sure that it meets the user's requirements. The main features of FUSION are:

- it generates an optimal execution plan automatically from the abstract requirements that a user may specify,
- it verifies that the result of execution meets the user's requirements, and
- it recovers from an execution plan if it does not meet the requirements of the user.

2.3.9 Proteus

Proteus [GKP$^+$03] is a system designed for dynamically composing and executing plans that integrate Web services in the presence of failure and Web services migrations. It monitors and shows the status of the composed components at run time. Proteus is a typical example of a wrapper-based composition system. It converts online sources into Web services and automatically composes XML Web services by building wrappers around the services. The main features of Proteus are [AES06]:

- it designs efficient execution plans by minimizing the number of Web services invoked and focusing on the efficient transmission of XML files, and
- it uses visualization tools for monitoring the execution of the plans.

2.3.10 SPACE

The *Structure Process Analyze based Composition Environment (SPACE)* [JWY09] is an architecture for defining, describing and evaluating service compositions formally. It defines the service composition based on situation calculus language. SPACE estimates the similarity of services by the basic structures and constraints rather than the features of single service. SPACE provides a formula to evaluate similarity of original and changed environment. It helps to find non-optimal but acceptable substitutes for services and guarantee the verifications of service composition. SPACE can provide a solution to choose a substitute for unavailable service.

2.3.11 StarWSCop

Star Web Services Composition Platform (StarWSCoP) [SWZZ03] is a dynamic Web service composition platform. StarWSCoP is developed on the Web services platform StarWS. StarWSCop includes several parts:

- *Intelligent System*: It decomposes the user's requirements into simple tasks and sequences them into abstract services.
- *Service Registry*: It is used to register Web services that can be discovered by the *Service Discovery Engine*.
- *Service Discovery Engine*: It looks up services in the registry and selects the service that satisfies the requested requirements.
- *Composition Engine*: It schedules the Web services composition in order. It keeps the composition trace information in the *Service Execution Information Library*. It also deals with events sent by the *Event Monitors*.
- *Wrapper*: It is used to hide the details of Web services and to achieve interoperability.
- *Service Execution Information Library*: It is used to store the composition trace information.
- *QoS Estimation*: It estimates the real-time QoS metrics of current composite Web services.
- *Event Monitors*: They are used to monitor various events and notify the Composition Engine.

StarWSCop extends WSDL with QoS attributed such as time, cost or reliability. This extension is used to support QoS based dynamic Web services composition. To achieve semantic match for Web services, an ontology-based layer is added.

2.3.12 METEOR-S

The *Managing End-To-End OpeRations for Semantic Web services (METEOR-S)* [VGS+05] [AVMM04] is a platform for the execution and enactment of semantic Web services and processes developed by the University of Georgia. It uses semantics for the complete life-cycle of semantic Web services. It adds semantics to current industry standards such as WSDL.

The services are described in *SAWSDL* (Semantic Annotation for WSDL). *SAWSDL* is a simple extension of WSDL. It provides a mechanism to connect the capabilities and requirements of Web services defined in WSDL with semantic concepts defined in an external ontology.

METEOR-S uses QoS ontologies to represent the semantic of the services nonfunctional properties. The supported nonfunctional properties are: time, cost, reliability and fidelity.

METEOR-S supports static and dynamic compositions. Its engine is limited to automatic binding of services according to user-defined constraints. The user manually provides an abstract process containing restrictions on services for each activity of the process. Then, these abstract processes are transformed to real processes by binding each activity to a real service at runtime. However, this capability is limited. The general structure of the process cannot be changed. For example, it is not possible to replace two activities by a service which matches the sum of two constraints instead of each of the constraints alone [KM05].

2.3.13 SeCSE

The *Service-Centric Systems Engineering (SeCSE)* [PBS+09] project is concerned with creating new methods, tools and techniques for requirements analysts, system integrators and service provision. It supports the cost effective development and use of dependable services and service-centric applications.

The discovery of services in SeCSE is based on the user service request defined using UML Use Case specifications. The functional requirements of the user request in defined using VOLERE. The Use cases and the functional requirements are expressed in natural language using the tool UCaRE. The service request is passed to the SeCSE service discovery engine (EDDiE). The matching services are then presentenced to the user.

The service composition in SeCSE is semi-automatic. The service composition is specified depending on a workflow defined by the user of the system. The actual service composition is performed based on the list of services automatically discovered from the service requests. The user can also specify binding rules that allow a dynamic adaptation of the service composition according to the defined constraints.

The service composition and the dynamic binding rules are specified during design-time. The dynamic adaptations of the service composition are performed at runtime in cases where some events occur such as a failure or unavailability of a service.

2.3.14 DynamiCoS

The *Dynamic Composition of Services (DynamiCoS)* [SPvS09] is a framework that aims to support automated service composition at runtime. DynamiCoS defines ontologies as a domain conceptualization. Service developers publish their semantic based services in the framework. The services semantic descriptions have to refer to the framework's ontologies. DynamiCoS consists of the following components:

- **Service creation module**: It is responsible for creating the services semantically, in terms of inputs, outputs, preconditions, effects, goals and nonfunctional properties, using the framework domain ontologies' semantic concepts.

- **Service publication module**: It is responsible for the publication of services described in different languages.

- **Service request module**: It is responsible for handling users' requests from different interfaces.

- **Service discovery module**: It is responsible for discovering the services that matches the users request semantically.

- **Service composition module**: It is responsible for handling the service composition following the graph composition algorithm.

2.3.15 TSCN

In [FYCL09], the authors propose a *Trust Service Composition Net (TSCN)* model based on Petri nets. They use TSCN to simulate the process of service composition. TSCN is also used to model services, components, the relationships between components and the operation mechanism of service composition. TSCN adopts the concepts of Trust matrix which represent the relationships between state, and dynamic trustworthy service composition strategy. The authors also propose a method to enforce trustworthiness.

The TSCN trustworthiness is evaluated using attached credibility to each service. Using those values the TSCN model can choose dynamically the service that best achieves the required credibility. TSCN does not support dynamic service composition planning.

2.3.16 Analysis

The service provision frameworks discussed above have been compared with respect to the following criteria. Except for context, formalism, and trustworthiness all other criteria are different from the criteria used earlier to compare service modeling and service compositions. Moreover, this list of criteria should be understood in order to appreciate the merits of the *FrSeC* framework discussed in this thesis.

- *Dynamic Selection:* The service provision framework should be designed to allow service requesters specify the requirements with the full knowledge that some service bindings may occur only at run time.

- *Dynamic Composition:* With the increased number of services and the increased composition complexity, it is difficult to have all service compositions predefined in a static manner.

- *Context Support:* Contextual information is essential at service publication, service query, service selection and planting, and service execution.

- *Semantic Support:* Semantic information is essential at service specification, service query, and service composition.

- *Formal:* Formalism is necessary to 1) verify the interaction between services by making sure there are no incompatible behaviors between services in a composition, 2) achieve correct automatic composition by verifying that the composition satisfies the requirements of the requester, and 3) check the conformance of requester requirements and the contracts of the services being provided.

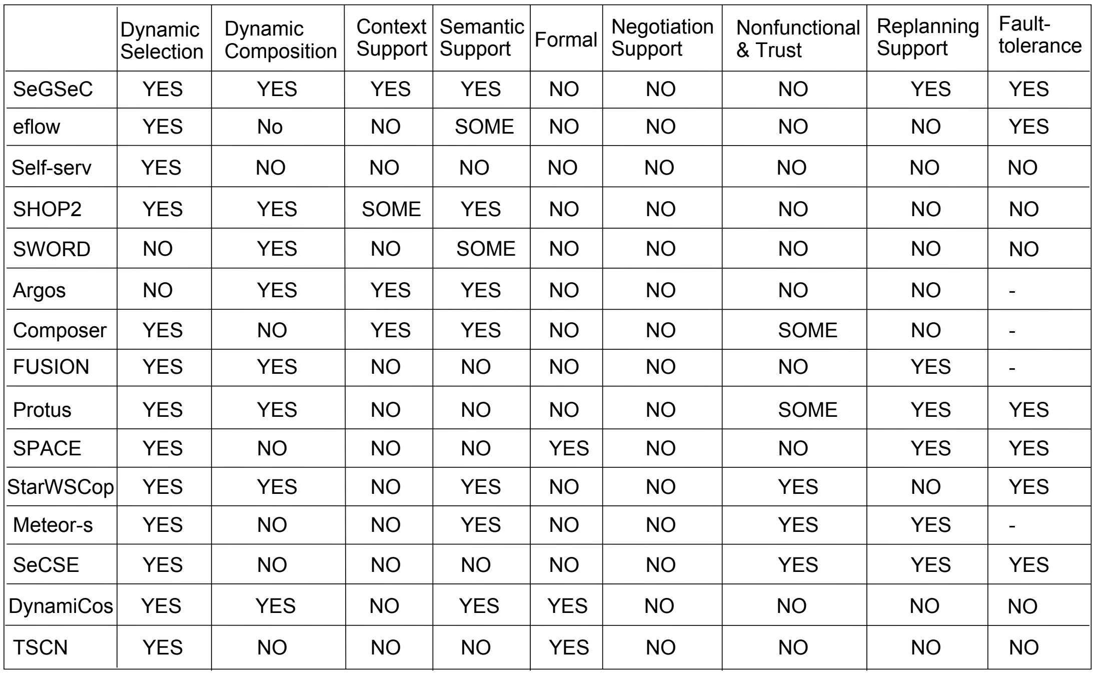
- *Negotiation Support:* Each service requester has his own set of requirements. In many cases, none of the available services may fully match these requirements. The service provision framework should provide a mechanism to support the negotiation between service requesters and providers.

- *Nonfunctional and Trust:* The consideration of nonfunctional and trustworthiness properties in service publication, discovery and ranking is essential.

- *Replanning Support:* At run time, the contextual information of the service consumer and requester might change. The service provision framework should support a replanning process to generate a new plan that best satisfies the requirements in the new context.

- *Fault-tolerance:* If a service fails or becomes unavailable at run time, the service provision framework should recover from this failure by selecting alternative services.

The result of this comparison is presented in Figure 4 and it shows the following:



	Dynamic Selection	Dynamic Composition	Context Support	Semantic Support	Formal	Negotiation Support	Nonfunctional & Trust	Replanning Support	Fault-tolerance
SeGSeC	YES	YES	YES	YES	NO	NO	NO	YES	YES
eflow	YES	No	NO	SOME	NO	NO	NO	NO	YES
Self-serv	YES	NO	NO	NO	NO	NO	NO	NO	NO
SHOP2	YES	YES	SOME	YES	NO	NO	NO	NO	NO
SWORD	NO	YES	NO	SOME	NO	NO	NO	NO	NO
Argos	NO	YES	YES	YES	NO	NO	NO	NO	-
Composer	YES	NO	YES	YES	NO	NO	SOME	NO	-
FUSION	YES	YES	NO	NO	NO	NO	NO	YES	-
Protus	YES	YES	NO	NO	NO	NO	SOME	YES	YES
SPACE	YES	NO	NO	NO	YES	NO	NO	YES	YES
StarWSCop	YES	YES	NO	YES	NO	NO	YES	NO	YES
Meteor-s	YES	NO	NO	YES	NO	NO	YES	YES	-
SeCSE	YES	NO	NO	NO	NO	NO	YES	YES	YES
DynamiCos	YES	YES	NO	YES	YES	NO	NO	NO	NO
TSCN	YES	NO	NO	NO	YES	NO	NO	NO	NO

Figure 4: Comparison of Service Provision Frameworks

1. With the exception of SWORD and Argos, all approaches support dynamic selection.

2. Dynamic composition is considered by almost half of the approaches. In most of these approaches AI planning techniques are used.

3. Contextual information is used by very few approaches. In these approaches, context is used to filter the services not to constrain the service contract. Hence, the relationship between the contract and context is not considered.

4. Semantic information using ontology is supported by almost half of the approaches. The use of ontology restrains the semantic support due to the complexity and difficulty of composing ontologies.

5. With the exception of three approaches, all remaining approaches are not formally based. This will limit their verification support.

6. None of the investigated approaches supports negotiation.

7. The support of nonfunctional and trustworthiness properties is very simple and limited.

8. Replanning is supported by almost half of the approaches. With the exception of protus, approaches that support replanning do not support dynamic composition and hence the replanning is manually performed.

9. Fault-tolerance is supported by only few approaches. A number of approaches such as Argos, Composer, FUSION and Meteor-s do not mention fault-tolerance. Hence, by default we consider that they do not support fault tolerance.

2.4 Summary

This chapter has presented a review and a comparison of the work most relevant to the goals of this thesis. The findings of the comparison reveal that there is a great need to innovate and add features to the current state of the art of SOC. The rest of this thesis deals with specific contributions made towards this goal.

Chapter 3

Research Methodology

This chapter presents the objectives of this thesis research and shows the different research steps, which together formulate the research methodology used to reach the objectives.

3.1 Research Objectives

The goal of this thesis is to present a formal approach for the development of trustworthy context-dependent service-oriented systems. The objective is achieved by putting together the following contributions:

- A formal service model for the specification of trustworthy context-dependent services.
- A formal composition and verification approach for trustworthy context-dependent services.
- A formal framework for the provision of trustworthy context-dependent services.
- A set of languages to support the provision of trustworthy context-dependent services.

3.2 Research Methodology

The research methodology is divided into four phases. The first phase is concerned with defining a formal model for trustworthy context-dependent services. The second phase is concerned with defining the composition and verification approach for trustworthy context-dependent services. The third phase is concerned with defining the service provision framework and all its elements. Finally, the fourth phase is concerned with introducing the set of languages to supports the provision of trustworthy context-dependent services. The following four subsections describe the research problems, research questions and the solutions provided by this thesis for the stated problems.

3.2.1 Phase 1: Defining a Formal Service Model

This section presents the research problems in defining a formal service model for trustworthy context-dependent services. There are four research problems outlined in this section. For each problem, we discuss the corresponding challenging research questions and provide our proposed solution.

Research Problem 1-A: Providing support for trustworthiness information

Problem Statement: Current service models focus only on specifying the functional behavior of services. They provide only limited or no support for specifying trustworthiness properties. Therefore, these models do not fit the need to define a trustworthy service. In order to provide services that meet the trustworthiness requirements of service requesters, there is a need to extend service definitions with specification of trustworthiness properties.

Research Questions: Below we discuss the questions related to the research problem 1-A and provide solutions.

Q1: ***What are the essential trustworthiness properties?*** We define trustworthiness properties to be expressed in two parts. One part is *ServiceTrust* in which service providers lists the quality claims in service provision. The features safety, security, availability, and reliability are included here. Safety means that no damage will happen during transit or communication and timeliness is guaranteed. Security is a composite of data integrity and confidentiality. Availability is specified as the maximum time of repair until the service returns back to operate correctly. Reliability is defined as the guaranteed maximum number of failures in a unit of time. In services that involve the delivery of tangible products, such as the vehicle delivered in car rental service, this section will include trustworthiness properties of the product being delivered. The second part is *ProviderTrust* which is the trust that consumers have on the service provider. It includes recommendations from other clients, and peer groups.

Q2: ***How to specify trustworthiness properties?*** We specified trustworthiness properties pertaining to quality claims in first order predicate logic. This will enable the properties to be verified formally. Although there is no agreed upon definition for ProviderTrust, our formalism allows the inclusion of any verifiable information.

Research Problem 1-B: Binding context to the service contract.

Problem Statement: Current service models provide limited or no support for specifying contextual information. Therefore, they ignore the relationship between service contracts and context. In order to provide services that meet the contextual requirements of service requesters, there is a need to extend service definitions with specification of contextual information.

Research Questions: Below we discuss the questions related to the research problem 1-B and provide solutions.

Q3: ***What type of context is essential in service definition?*** We believe that three types of contextual information are necessary in service provision. The first is the contextual information of the service provider. The second is the contextual information of the service requester. The third is the service execution context. Service requester context should meet the context constraints defined as part of the service. Service provider context will affect the trust level of the service consumer toward the service provider. The execution context will ensure that the contextual constraints are not violated during service execution.

Q4: ***How context can be specified?*** We used the context definition of Wan [Wan06] where context is defined as a pair of tags and values. Mostly the dimensions related to WHO, WHERE, WHEN, WHY and WHAT seem sufficient.

Research Problem 1-C: Including the legal rules in service definition

Problem Statement: Current service models provide limited or no support for specifying legal rules. There are a few approaches that specify legal rules informally, but do not distinguish them from nonfunctional properties. It is necessary to distinguish between legal rules and nonfunctional properties because the former is part of a contract and changeable, whereas the later is relevant only to service functionality which does not change. Another important reason is that a violation of a nonfunctional property might be acceptable by the client, while violating a legal rule might result in contract termination. Therefore, there is a need to extend service definitions with specification of legal rules separately from nonfunctional properties.

Research Questions: Below we discuss the questions related to the research problem 1-C and provide solutions.

Q5: ***What type of legal rules are essential in service definition?*** We believe that business rules and trade laws that are enforced at service provision and service delivery are necessary in service definitions. Business policies governing refund, administrative charges, penalties, and service requester's rights are essential in a contract.

Q6: ***How legal rules can be specified?*** We used predicate logic to specify legal rules in the contract part of the service model. The use of logic will enable formal verification.

Research Problem 1-D: The need for a service model for trustworthy context-dependent services

Problem Statement: Current service models provide only limited or no support for specifying legal rules, trustworthiness properties, and contextual information. Hence, current approaches cannot be used to specify trustworthy context-dependent services. Moreover, current service models are only informal, and consequently no analysis on the stated claims can be done rigorously. Therefore, there is a need for a new service model that is formal, and rich in structure to specify trustworthy context-dependent services.

Research Questions: Below we discuss the questions related to the research problem 1-D and provide solutions.

Q7: ***What is the structure of service model and how it is formalized?*** We packaged the service functionality and the attributes necessary to execute it in the ‘service part’ of the model. We packaged the trustworthiness claims, legal rules, and context information in the ‘contract part’ of the model. Thus, the new service model has the ‘service part’ and the ‘contract part’. The service model is formalized using set theory and logic. The formalized service model is hidden from users, and is made available for analyses purposes. An informal, yet precise, description of the service model is published.

Q8: ***What type of analysis is required on the new service model?*** We defined three types of analysis on our services. The first is performed before publication and is usually the responsibility of service providers. The second is performed before execution and is usually the responsibility of the unit responsible of execution, or service providers. The third is performed after service delivery and is done by service providers and requesters. The goal of the third type is to ensure that both parties satisfy their contractual obligations.

3.2.2 Phase 2: Defining a Service Composition Theory

There are two research problems outlined in this section. For each problem, we discuss the corresponding challenging research questions and provide our proposed solution.

Research Problem 2-A: The lack of a composition theory for trustworthy context-dependent services.

Problem Statement: Current service composition approaches are ad-hoc, and do not have a semantic basis. Moreover, these composition methods are like functional compositions because legal rules, trustworthiness properties, and nonfunctional properties are omitted in the composition. Hence, there is a need for developing a composition theory for trustworthy context-dependent services.

Research Questions: Below we discuss the questions related to the research problem 2-A and provide solutions.

Q9: ***How to compose trustworthy context-dependent services?*** The service model is formal, based on model-based specification theory. As such, service models are composable. We have introduced composition operators and provided their formal semantics. Therefore, a service composition becomes an expression involving service models and composition operators. The meaning of an expression is uniquely obtained by applying the semantics of composition operators, left to right, to the service expression. If ∇ is a composition operator and S_1, and S_2 are two services, then the composition $S_1 \nabla S_2$ is obtained by composing $service\_part(S_1)$ ∇ $service\_part(S_2)$ and $contract\_part(S_1)$ ∇ $contract\_part(S_2)$ where the business logic is appropriately chosen for these two separate compositions. Consequently, we do the following:

- To compute $service\_part(S_1) \nabla service\_part(S_2)$: The functionalities of S_1 and S_2 are composed, the attributes of S_1 and S_2 are composed, and the nonfunctional properties of S_1 and S_2 are composed. The semantics of each of the individual compositions is driven by the business logic of the service provider, yet governed by the underlying data type semantics.

- To compute $contract\_part(S_1) \nabla contract\_part(S_2)$: The trustworthiness properties in S_1 and the trustworthiness properties in S_2 are composed, the legal rules in S_1 are composed with the legal rules in S_2, and finally the context information in S_1 is composed with the context information in S_2. These individual compositions are driven by the business logic of the service provider, yet governed by the underlying predicate logic and context semantics.

Research Problem 2-E: Formal verification approach for the verification of composite trustworthy context-dependent services.

Problem Statement: Current verification approaches can only verify the functional behavior. Often, it is done only informally. Verifying a composite service for a specific property, as defined in our model, is a complex task. Hence, there is a need for a formal verification approach.

Research Questions: Below we discuss the questions related to the research problem 2-E and provide solutions.

Q14: ***How to verify a property in a composite service?*** We use a model transformation approach. This approach transforms a service composition into a UPPAAL timed automaton, and then uses the UPPAAL model checking tool to verify the property in the automaton.

3.2.3 Phase 3: Defining a Service Provision Framework

This section presents the research problems in defining a formal service provision framework for trustworthy context-dependent services. There is one research problem outlined in this section. For this problem, we discuss the corresponding challenging research questions and provide our proposed solution.

Research Problem 3-A: Designing a formal service provision framework for trustworthy context-dependent services.

Problem Statement: Current service provision approaches don't consider legal rules, trustworthiness properties and context information is service publication, discovery and provision. Hence, there is a need for a service provision framework for providing trustworthy context-dependent services.

Research Questions: Below we discuss the questions related to the research problem 3-A and provide solutions.

Q15: ***What are the essential features of the service provision framework?*** We believe that the essential features are (1) context gathering, and analysis, (2) support for trusted transactions, (3) support for nonfunctional and trustworthiness properties, (4) support for dynamic selection and planning, (5) semantic support, (6) fault-tolerance support, (7) formalism, and (8) replanning and negotiation support.

Q16: ***What are the essential elements of the service provision framework?*** We believe that the essential elements for the service provision framework are (1) service registry, (2) planning unit, (3) context gathering unit, (4) execution unit, (5) plan negotiation unit, (6) service provider unit, (7) trusted authority unit, and (8) service requester unit.

Q17: ***How to ensure trustworthy transactions in the service provision framework?*** We controlled user access to the framework components by requiring authentication certificates for every session of activity. The trusted authority is responsible for providing service providers and requesters with such certificates.

Q18: ***How services can be stored with semantic information?*** We defined the registry to support semantic information. It is structured in a specific manner. It is divided into domains and sub-domains. It is built with the input from service providers.

Q19: ***What type of queries does the framework need to support?*** Service requesters have different needs and they may have differing technical expertise. So, we defined two query types. One is the traditional query type which requires users to specify their requirements, and the other is the buffet type where users buy what they see in the service registry. The traditional query itself may be formulated with weights, which specify the extent of desirability of a required feature of service.

Q20: ***How to rank services in case of multiple matches to the requester query?*** Often, the requirements of the service requester can be met fully or partially by multiple services. In such cases, a ranking of the candidate services is necessary. The ranking will be performed according to a ranking algorithm that considers all the requirements and the priorities of the services requester. Current ranking algorithms consider functional properties and few nonfunctional properties in the ranking algorithm. There is no approach that considers trustworthiness properties, legal rules and context information. Hence, we introduced a new ranking algorithm that considers all properties.

Q21: ***How to enable flexible contracts?*** We introduced a new element that is responsible for managing negotiations between service providers and requesters. We also introduced a methodology to extend service contracts.

3.2.4 Phase 4: Defining the Languages to Support the Service Provision Framework

This section presents the research problems in defining the languages to support the formal service provision framework for trustworthy context-dependent services. There is one research problem discussed in this section. For this problem, we discuss the corresponding challenging research questions and provide our proposed solution.

Research Problem 4-A: The need for language support.

Problem Statement: Formal specification of service definitions are required for formal analysis. A service provider may not have the background to create a formal specification. Formulating service queries should also be made simple in order to enable all clients to feel at ease in querying for services. Therefore, there is a need for easy to use languages to support the publication and discovery of trustworthy context-dependent services.

Research Questions: Below we discuss the questions related to the research problem 4-B and provide solutions.

Q22: ***What are the necessary languages for the publication of services?*** Current service definition languages supports only functional and some nonfunctional properties. Hence, there is a need for new languages to support our service model. We defined a set of languages that follow our service model. Service providers use these languages to publish their services. The main language is CSL which is a textual language. The second language is CSDL which is XML-based. Service providers do not need to worry about complicated formal methods.

Q23: ***What are the necessary languages for querying services by service requesters?*** Current query languages use only functional properties. Hence, there is a need for a new set of languages that support our rich service model. We introduced two languages. The first language is SQL which can be used by service requesters to formulate the two types of queries. The second language is CSQL which is XML-based, which is used for exchanging queries between framework elements.

Q24: ***What are the necessary languages to support the provision framework interactions?*** We created the family Service Processing languages (SPL). That is, SPL is a package of languages that can be used for (1) the specification of the elements of the framework, and (2) the specification of the messages and data transferred between the elements. SPSL family includes the languages Service Registry Language (SRL), Service Query Language (SQL), Trusted Authority Language (TAL), Service Negotiation Unit Language (NUL), and Service Planning Unit Language (SUL). SRL is used to specify the elements of the service registry. SQL is used to specify queries by service requesters. TAL is used to request authentications and analysis from the trusted authority. NUL is used to specify negotiation requests. SUL is used to specify the requests sent to the registry and specify the planning results. With basic exposure to abstract data types and logi, SPL specifications can be easily written and understood.

Q25: ***What are the necessary languages to support the exchange of information between the provision framework elements?*** The elements in the architecture will exchange rich information, such as entire query structure or the certificate. The languages for this purpose must have 'interoperability' property. So, we created five XML-based languages to be used by the framework elements for exchanging information. These languages are mapped to SPL languages. The XML version of CSL is called ConfiguredService Description Language (CSDL), the XML version of SQL is called ConfiguredService Query Language (CSQL), the XML version of NUL is called Negotiation Unit Description Language (NUDL), the XML version of TAL is called Trusted Authority Description Language (TADDL), and the XML version of SUL is called Planning Unit Description Language (SUDL). These languages are faithful XML translations of their respective languages.

3.3 Summary

This chapter has presented the research methodology followed in this thesis. Research problems pertaining to our research goals have been raised and their solutions outlined. Currently, to the best of our knowledge there exists no work in the area of SOC which has raised a wide range of issues such as the ones we are solving in this thesis.

Chapter 4

ConfiguredService Formalism

This chapter introduces two important concepts, namely *ConfiguredService* as a higher-order data type, and *flexible service contract* as a formal template. After developing a concrete syntax for *ConfiguredService* its expressive power is brought out with several examples, in particular with the formal representation of the case study 'Auto Roadside Emergency Service'. Set theory and logic, the basic mathematical toolkit for model-based specification approach, have been used to formalize *ConfiguredService*. The properties *completeness of functionality*, *context compatibility for service provision and service delivery*, and *contract consistency* are identified as essential in *ConfiguredService* representation and an explanation of formally analyzing a *ConfiguredService* for these properties are discussed. The main purpose behind flexible service contract template is to provide a formal negotiated meeting ground between a service provider and its service requester.

4.1 ConfiguredService Type

In Chapter 2, a number of approaches for modeling services have been compared. This comparative study has revealed the following deficiencies in the current approaches.

- lack of a service model that collectively considers the relationship between contract and service,

- absence of contextual information in service description,
- blurred boundary between nonfunctional and trustworthiness properties,
- disregard of legal rules, and
- lack of formalism.

The service structure, called *ConfiguredService*, puts forth in this Section tightly couples service and contract. Functionality of service, data related to service, service attributes, and nonfunctional properties are grouped under ‘service’. Legal rules, trustworthiness claims, and context information are grouped under ‘contract’. As will be explained below, this definition is both rich in structure and precise in its semantics. Consequently, this service definition will enable a better publication and discovery of services, and will increase the user trust in services.

4.1.1 Rationale for ConfiguredService Definition

In traditional transaction based systems, the term ‘service’ refers to the functionality and the behavior projected out of certain system operations. Examples include ‘update services in database systems’, and ‘credit checking services’ in financial auditing systems. In Service-oriented Computing, service is a first class object and consequently there is need to come up with a richer meaning for services. We should capture in its definition ‘what a service is’ and ‘what requirements are to be met for providing it’. Motivated by this need we define a *ConfiguredService* to consist of the two parts *Service* and *Contract*.

We can think of the ‘service part’ to define the functional behavior of the service, as intended by its specification. This functionality is tightly related to a set of nonfunctional properties, such as the cost of invoking this functionality. This tight coupling motivated us to define a service to include (1) functionality, (2) nonfunctional properties, and (3) attributes. We can think of the ‘contract part’ as defining ‘the terms and conditions’ pertaining to service delivery. Typically these include context information on service availability and service delivery, legal (business) rules, and trustworthiness properties of the service. The information contained in the 'service part' is static. That is, the service functionality, the attributes related to the service (and its provider), and the nonfunctional properties associated with the service do not depend upon the changeable contractual entities, which include context, legal rules, and trustworthiness properties.

An important consequence of this definition is that the service description is loosely coupled with a contract description. By changing the contract part alone we can create many *ConfiguredServices*, all providing the same service. For example, providing a wireless Internet connection that costs 5$ per hour is a single service. This service might be associated with one contract stating that the quality of reception is *excellent*, provided the service requester is located within 50 meters from the base station. The same service may be associated with another contract stating that the quality of reception is *good*, provided the service requester is located beyond 50 meters but within 100 meters from the station base. Thus, we have two *ConfiguredServices* containing the same service but with different contracts.

The contract part in a *ConfiguredService* is given a rich structure to state the quality claims, legal bindings, and contextual constraints. The quality claims of the service provider are listed under trustworthiness section. Trustworthiness properties include safety, security, reliability, and availability guarantees of the service provider, and peer and client recommendations related to the service, awarded to the service provider. Trustworthiness properties influence significantly the consumer's intent to buy the service. Hence they must be represented with some formality and precision in order to be able to devise efficient selection procedures for service discovery. We emphasize the reason for separating nonfunctional properties, defined as part of the service, from trustworthiness properties, defined as part of the contract. The nonfunctional properties represent the information that are static for a service and can be quantified. An example is the cost of the service. Trustworthiness properties are claims made by the service provider on the quality of the service. They cannot always be quantified, however they should be verifiable. Trustworthiness properties are defined as logical expressions in order to enable formal verification.

It may be argued that the cost of service might change according to the service requester, and cost, being 'dynamic', it should be part of the contract section. Our response to this argument is that the price itself is fixed, however under some exceptions discounts might be offered. These arise from business policies and hence it is appropriate to list them in *Legal Issues* section of the *Contract*. For example, a discount offered to senior citizens is a business rule which constrain the price with respect to age. In general, a legal rule is a business rule constraining the execution of service. Legal rules may also include rules governing exceptions, exclusions, contract violation, renewal, and termination. The *Legal Issues* section is the most important dynamic aspect of the contract, especially for services offered globally. The trust guarantees enshrined in the contract and the legal rules themselves are not absolute. They depend upon certain contextual constraints, such as the domicile of the consumer. For example, a shipping service might guarantee a next day delivery. But this guarantee may be conditioned by the shipping location. The location where service is to be delivered might bring in local laws to be included in the legal rules of the contract. For example, sales tax might be exempt in certain regions, whereas in some other regions a value-added tax may have to be added on top of the sales tax imposed on the service by the service provider. The information that constrain the service contract is defined as *context*. Context is any type of information used to characterize an object or situation [Dey01]. We use the notation from [Wan06] to define context and a logical expression to state a *context rule*. A context rule is a situation which might be true in some contexts and false in some others. For example, the situation $WARM = Temp > 30 \ \wedge \ Humid > 70$, is true only in contexts where the temperature is greater than 30 degrees, and the humidity is greater than 70. Such a situation may have to be validated in order to provide context-dependent 'heating service'.

A *ConfiguredService* description can be formalized. The functionality of service can be specified by a precondition and a postcondition, which are first order logical expressions. Trustworthiness properties, legal rules, and situations can all be specified as logical expressions. The data and attribute in the *ConfiguredService* should be sufficiently complete in order to enable a formal validation of the entire *ConfiguredService* description. We discuss in Section 4.3 three verification scenarios. The service specified in a *ConfiguredService* can

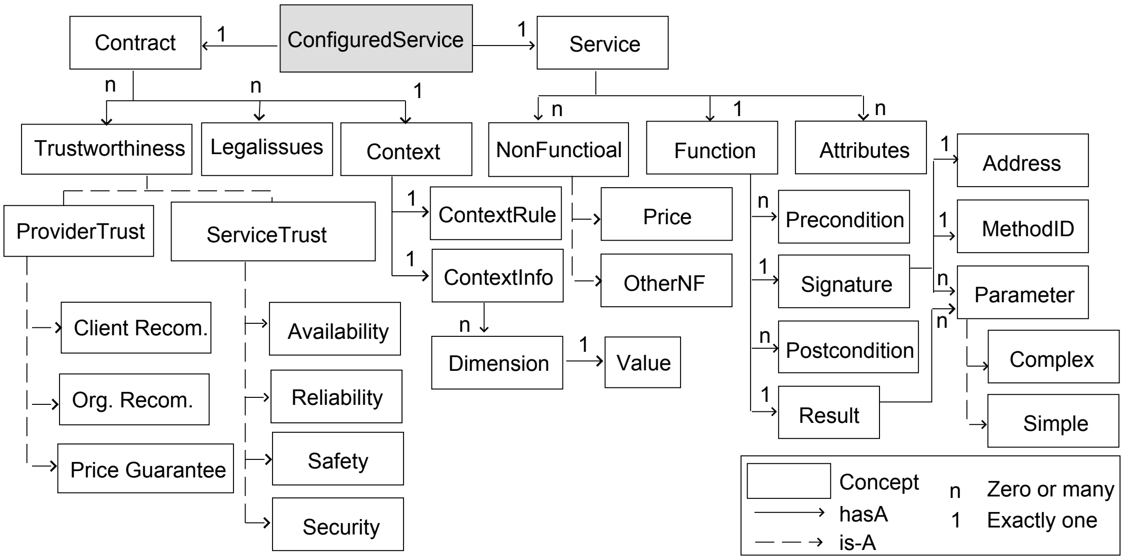


Figure 5: *ConfiguredService* Structure

be provided only if the three step evaluation is successful.

4.1.2 Informal Semantics of ConfiguredService

A *ConfiguredService* is divided into the two main parts *Service* and *Contract* as shown in Figure 5. Below, an informal description of the elements in these two parts of a *ConfiguredService* is given.

Service Description

The *Service* section has the three parts *Functionality*, *Nonfunctional properties* and *Attributes*.

1. *Functionality*: Its definition includes the function *signature*, *result*, *precondition* and *postcondition*. The *signature* part defines the function *identifier*, the invocation *address*, and the *parameters* of the function. The function invocation has the same effect as in a programming environment, since service function is an autonomous program. Each parameter has an *identifier* and a *type*. The *result* part defines the returned data of the service function. The *precondition* should be made true, either

by the service provider or the consumer, in order to make the function available. The *postcondition* is guaranteed by the service provider to be true after service execution.

2. *Nonfunctional properties*: The nonfunctional properties associated with the service are listed in this section. Pricing information, which can itself be a complex property expressing different prices for different amount of buying, is an example of nonfunctional property. For some types of services, such as video downloading, the amount of storage required and speed of downloading may be included as nonfunctional properties.

3. *Attributes*: Every attribute is a type-value pair. Attributes provide sufficient information that is unique to a service. As an example, for selling a book the appropriate attributes are title of book, author name, year of publication, and publisher information.

Contract

The *Contract* is divided into the three main parts *Trustworthiness*, *Legal Issues* and *Context*.

1. *Trustworthiness:* Trustworthiness properties are expressed in the two sections *ServiceTrust* and *ProviderTrust*.

   ServiceTrust section lists the trustworthiness claims that the service provider makes on the product and service. Following the definition of trustworthiness [MA11], we have chosen to include in this section *safety*, *security*, *availability*, and *reliability* claims of the service provider on the service. Safety means timeliness guarantee and an assurance that no damage will happen during service transit and delivery. Security is a composite of data integrity and confidentiality. Data integrity is concerned with the techniques to ensure the correctness of data after communication. Data confidentiality is concerned with the privacy of data during communication. Availability and reliability are defined in [MA11] in terms of failures and repairs. A failure is defined as a deviation from the correct service behavior. A repair is defined as a change from incorrect service to correct service. Hence, availability is specified as the maximum time of repair until the service returns back to operate correctly, and reliability is defined as the guaranteed maximum number of failures in a unit of time. In services that involve the delivery of tangible products this section will include trustworthiness properties of the product being delivered. As an example, for car rental service safety features of the vehicle and the history of its maintenance will be included in this section.

ProviderTrust is the trust that consumers have on the service provider. It includes recommendations from other clients, and peer groups. Although there is no agreed upon definition for *ProviderTrust*, we allow the inclusion of any verifiable trust recommendations of users and peers.

2. *Legal issues:* Business rules and trade laws that are enforced at the locations of service provision and service delivery are included in this section. Example policies govern *refund*, *administrative charges*, *penalties*, and *service requesters rights*. Such rules are expressible as logical expressions in predicate logic. Below is a sample set of legal rules.

    - Price conditions: These are rules that specify the categories of consumers who get to pay a discount on the regular price. An example rule is “a student with a valid ID gets a 15% discount on the regular announced price of service”.

    - Refund conditions: Refund policy allows a consumer to get either a full or partial amount of fee paid to get the service, in case of a contract violation. An unconditional refund policy allows a consumer to return the product or service within a stipulated period of time after the delivery of the product/service. A conditional refund policy may require a proof by the consumer that the service contract was violated.

    - Joining fee: These are rules that specify the categories of consumers who get to pay a joining fee and the fee amount. An example rule is “an activation fee of 35$ is required before getting a phone subscription”.

- Interest charges: These are rules that specify the penalties of late payments. An example rule is “if the full payment is not received on time, a 10% interest will be applied on the outstanding balance”.
- Administrative charges: These are rules specifying fees and penalties for violating payment conditions. An example rule “if a check is returned by the bank for lack of funds, a 25$ charge will be added to the payment”.
- Deposit rules: These are rules that specify the categories of consumers who are required to pay a deposit and the deposit amount. An example rule is “if the age of the driver is less than 25, a 300$ deposit is required before renting a car”.
- Payment Rules: Some typical rules related to the payment for a service are listed below:
  - Payment methods: These rules specify the accepted payment methods. An example rules is “only cash payments are accepted”.
  - Payment schedule: These are rules that specify the required payment schedule. An example rule is “the payment should be received on the first day of every month”.
  - Payment discounts: These are rules that specify the discounts to be awarded if a specific payment schedule or method is followed. An example rule is “a 12 months advance payment is eligible for a 25% discount”.
  - Payment fees: These are rules that specify extra fees associated with a specific payment method. An example rule is “for credit card payments a fee of 5% is added to the total amount”.
- Service requester’s penalties: These are rules that specify the penalties applied on the service requester for not complying with the service contract. An example rule is “if the contract is canceled before 12 months of subscription, an early termination fee of 200$ is applied”.
- Service provider’s penalties: These are rules that specify the penalties applied on the service provider for not complying with the service contract. An example

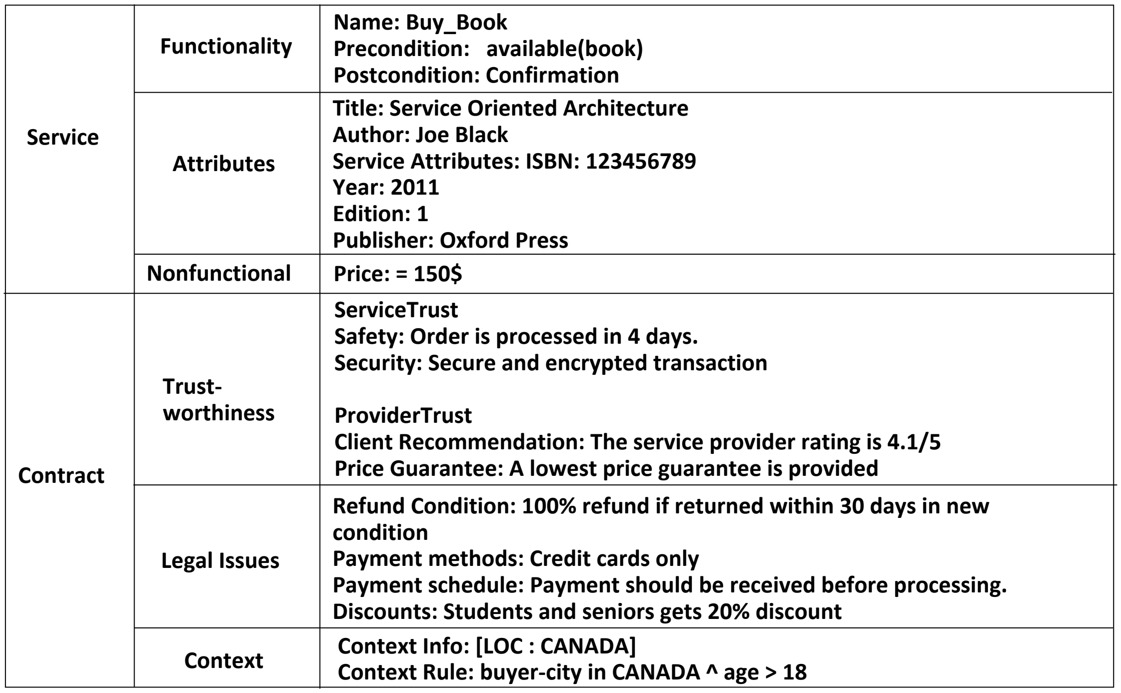
rule is “failure to deliver the service on time will result in the service being free”.

- Rights: These are rules that specify the service requesters’ rights. An example rule is “the service requester has a warranty on the product for one year”.
- Obligation Rules: These rules specify the obligations that should be respected by service requester to fulfill the contract. An example rules is “the rented car should not be driven cross the border”.

3. *Context:* The context part of the contract is divided into *context info* and *context rules*. The contextual information of the service provider is specified in the *context info* section. The situation or context rule that should be true for service delivery is specified in *context rules* section. It is the responsibility of the service requester to validate the *context info* for obtaining the service, and it is the responsibility of the service provider to validate the *context rules* at service delivery time.

4.1.3 Examples

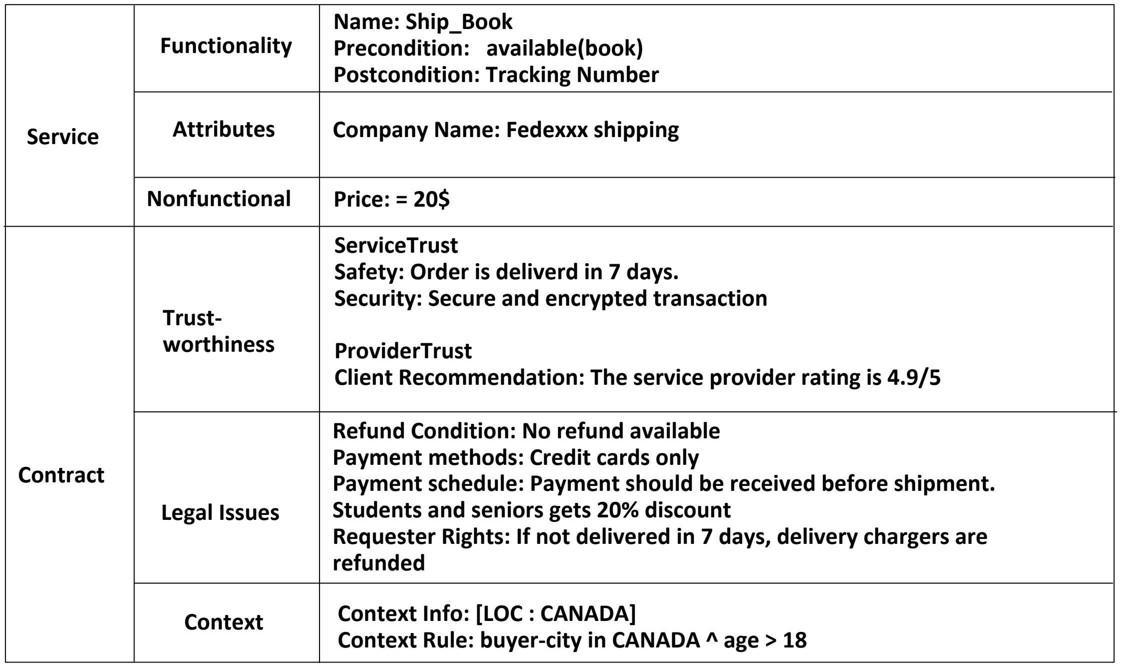
Before presenting *ConfiguredService* examples it is essential to highlight the difference between the terms product and service. When buying books online we are buying (physical/tangible) products. But, when buying life insurance or health insurance, or ordering cable service we are buying services. A product may provide many services. As an example, a cell phone may provide a ’calling’ service and a ’messaging’ service. Thus, a *ConfiguredService* might be of three types. The first type only involves a product. The second type only involves a service. The third type involves a product with services. Hence, in this section we present three examples corresponding to the three types of *ConfiguredService*. Table 1 describes the *ConfiguredService* for selling a book which is an example of a product. Table 2 shows the *ConfiguredService* for shipping books which is an example of a service. Table 3 describes the *ConfiguredService* for renting a car which is a product with services. In all examples, the *Functionality* section lists the function name, a precondition



Service	Functionality	Name: Buy_Book Precondition: available(book) Postcondition: Confirmation
	Attributes	Title: Service Oriented Architecture Author: Joe Black Service Attributes: ISBN: 123456789 Year: 2011 Edition: 1 Publisher: Oxford Press
	Nonfunctional	Price: = 150$
Contract	Trust-worthiness	ServiceTrust Safety: Order is processed in 4 days. Security: Secure and encrypted transaction ProviderTrust Client Recommendation: The service provider rating is 4.1/5 Price Guarantee: A lowest price guarantee is provided
	Legal Issues	Refund Condition: 100% refund if returned within 30 days in new condition Payment methods: Credit cards only Payment schedule: Payment should be received before processing. Discounts: Students and seniors gets 20% discount
	Context	Context Info: [LOC : CANADA] Context Rule: buyer-city in CANADA ^ age > 18

Table 1: Buy_Book Example

that must be true for service availability and a postcondition that must become true after service execution. The *Attribute* sections of the examples are strikingly different, because in the *Buy_Book* example a specific product type is to be described whereas in the *Ship_Book* and *Rent_Car* examples a generic service type is to be described. The *Contract* parts of the examples are similar, differing only in details. Trustworthiness rules are separated into claims made by the service provider, and the organizational and consumer recommendations. The former qualify the quality of ‘sell action’, ‘shipping action’ or ‘rent action’. It is important to highlight that in the *Rent_Car* example the trustworthiness properties included properties regarding the product itself, in this case the car. These rules are verifiable by inspection. The legal rules govern policies regarding refund conditions, payment issues, discount rules, and requester rights. The context rule must evaluate to true at service execution time. Thus, *ConfiguredServices* for the examples are *abstract data type* templates.



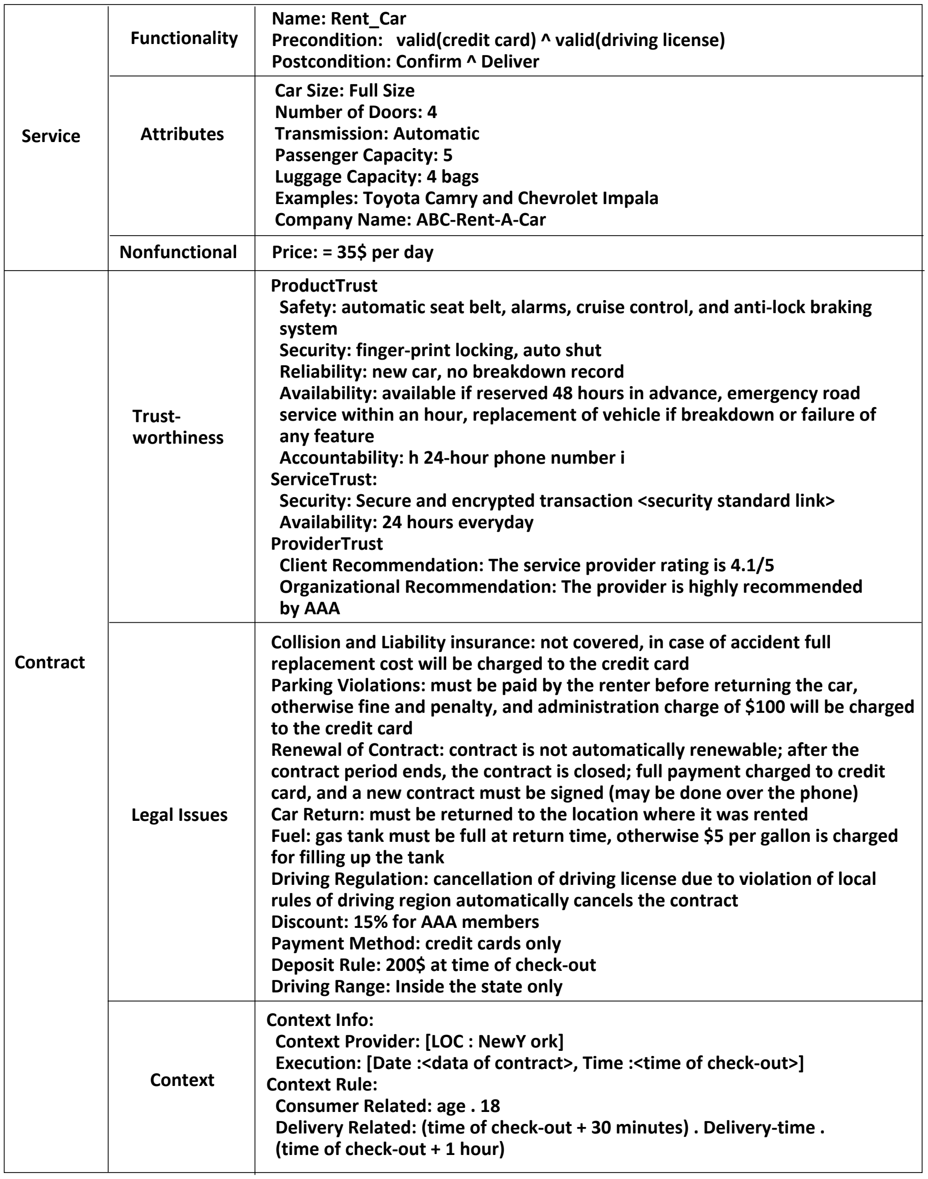
Service	Functionality	Name: Ship_Book Precondition: available(book) Postcondition: Tracking Number
	Attributes	Company Name: Fedexxx shipping
	Nonfunctional	Price: = 20$
Contract	Trust-worthiness	ServiceTrust Safety: Order is deliverd in 7 days. Security: Secure and encrypted transaction ProviderTrust Client Recommendation: The service provider rating is 4.9/5
	Legal Issues	Refund Condition: No refund available Payment methods: Credit cards only Payment schedule: Payment should be received before shipment. Students and seniors gets 20% discount Requester Rights: If not delivered in 7 days, delivery chargers are refunded
	Context	Context Info: [LOC : CANADA] Context Rule: buyer-city in CANADA ^ age > 18

Table 2: Ship_Book Example

4.2 Formal Representation of ConfiguredService

A *ConfiguredService* has both an informal and a formal description. The former is for publication by the service provider. All authorized system users, in particular service requesters, will have access to published services. The later is prepared by the service provider, *hidden* from the 'general clients', and is made available to the trusted authority that in turn will use it to analyze the claims made in the informal description of the *ConfiguredService*.

A *ConfiguredService* can be formally written using a model-based specification notation. For example, the service functionality can be written in predicate logic, the data section can be formalized as an aggregated abstract data type, the nonfunctional properties, trust attributes, and legal rules can be written as logical expressions. Thus, the contract part is a collection of logical formulas. The context information is written in the notation introduced by Wan [Wan06]. Because of this underlying formalism it is possible to rigorously verify the claims made in a *ConfiguredService*. Below we briefly outline the formal context notation and we give the formal representation of the rest of *ConfiguredService* elements.



Service	Functionality	Name: Rent_Car Precondition: valid(credit card) ^ valid(driving license) Postcondition: Confirm ^ Deliver
	Attributes	Car Size: Full Size Number of Doors: 4 Transmission: Automatic Passenger Capacity: 5 Luggage Capacity: 4 bags Examples: Toyota Camry and Chevrolet Impala Company Name: ABC-Rent-A-Car
	Nonfunctional	Price: = 35$ per day
Contract	Trust-worthiness	ProductTrust Safety: automatic seat belt, alarms, cruise control, and anti-lock braking system Security: finger-print locking, auto shut Reliability: new car, no breakdown record Availability: available if reserved 48 hours in advance, emergency road service within an hour, replacement of vehicle if breakdown or failure of any feature Accountability: h 24-hour phone number i ServiceTrust: Security: Secure and encrypted transaction <security standard link> Availability: 24 hours everyday ProviderTrust Client Recommendation: The service provider rating is 4.1/5 Organizational Recommendation: The provider is highly recommended by AAA
	Legal Issues	Collision and Liability insurance: not covered, in case of accident full replacement cost will be charged to the credit card Parking Violations: must be paid by the renter before returning the car, otherwise fine and penalty, and administration charge of $100 will be charged to the credit card Renewal of Contract: contract is not automatically renewable; after the contract period ends, the contract is closed; full payment charged to credit card, and a new contract must be signed (may be done over the phone) Car Return: must be returned to the location where it was rented Fuel: gas tank must be full at return time, otherwise $5 per gallon is charged for filling up the tank Driving Regulation: cancellation of driving license due to violation of local rules of driving region automatically cancels the contract Discount: 15% for AAA members Payment Method: credit cards only Deposit Rule: 200$ at time of check-out Driving Range: Inside the state only
	Context	Context Info: Context Provider: [LOC : NewY ork] Execution: [Date :<data of contract>, Time :<time of check-out>] Context Rule: Consumer Related: age . 18 Delivery Related: (time of check-out + 30 minutes) . Delivery-time . (time of check-out + 1 hour)

Table 3: Car_Rental Example

4.2.1 Context Formalism

Many of the rules must be validated in the context, as stated in the *Context* part of the *ConfiguredService*. As an example, the *car return* rule, shown in Table 3, is *intensional*, in the sense that it has a hidden context information which is the *location* where the car is rented. However, this information is available in *Context Info* part of the *ConfiguredService*. Hence, this rule must be evaluated in this context. Another example where context indirectly arises is in the rule *driving regulation* shown in Table 3. From the constraint *driving range*, it follows that all cities within the driving region are precisely the cities in the province in which the car is rented. Hence, the rule *driving regulation* is a set of rules, corresponding to the driving rules in the cities of the province in which the car is rented. It is clear that context must have a formal representation in order that evaluation of rules can be automated.

Context information is formally specified, as defined in [Wan06], using *dimensions* and *tags* along the dimensions. In our research, we have been using the five dimensions *WHERE*, *WHEN*. *WHAT*, *WHO*, and *WHY*. The dimension names are generic. For example, the dimension *WHERE* is associated with locality, the dimension *WHEN* is associated with temporal information such as time and date, the dimension *WHO* is associated with subjects (or roles), the dimension *WHAT* is associated with an activity, and the dimension *WHY* is associated with a purpose for the stated activity. In general, it is the responsibility of service providers to choose as many dimensions and their names in order to present the contexts associated with services. Assume that the service provider has invented a finite set $DIM = \{X_1, X_2, \ldots, X_n\}$ of dimensions, and associated with each dimension X_i a type τ_i. Following the formal aspects of context developed by Wan [Wan06], we define a context c as an aggregation of ordered pairs (X_j, v_j), where $X_j \in DIM$, and $v_j \in \tau_j$. As an example, the context in the *Context Info* part of Table 3 uses the three dimensions LOC (location information), $DATE$ (the day the car is rented) and $TIME$ (the time of checkout). With these dimensions a specific instance of the rental context can be written $c = [LOC : NewYork, DATE : 09/09/2011, TIME : 9]$. Thus, taken as a whole we get a multidimensional perspective of the car rental context.

A *context rule* is a situation which might be true in some contexts and false in some others. For example, the situation $VERYWARM = Temp > 40 \ \wedge \ Humid > 70$, is true only in contexts where the temperature is greater than 40 degrees, and the humidity is greater than 70.

4.2.2 A Model-based Formalization of other ConfiguredService Elements

After explaining the basic notation we build the model incrementally, according to the template in Figure 5. A constraint is a predicate logic expression, defined over data parameters and attributes. The UPPAAL verifier that is used in verifying the composition rules (Chapter 5) allows well-formed formula built by using standard logical operators, quantifiers, and temporal operators allowed in Timed Computation Tree Logic (TCTL) [BDL04a]. Therefore, in principle we could use the full power of TCTL to write the constraints in *ConfiguredService* description. However, all examples chosen involve only simple constraints.

Let $\mathbb{C}$ denote the set of all such logical expressions. $X \in \mathbb{C}$ is a constraint. The following notation is used in our definition:

- $\mathbb{T}$ denotes the set of all data types, including abstract data types.
- $Dt \in \mathbb{T}$ means Dt is a datatype.
- $v : Dt$ denotes that v is either constant or variable of type Dt.
- X_v is a constraint on v. If v is a constant then X_v is true.
- V_q denotes the set of values of data type q.
- $x :: \Delta$ denotes a logical expression $x \in \mathbb{C}$ defined over the set of parameters Λ. A parameter is a 3-tuple, defining a data type, a variable of that type, and a constraint on the values assumed by the variable. We denote the set of data parameters as $\Lambda = \{\lambda = (Dt, v, X_v) | Dt \in \mathbb{T}, v : Dt, X_v \in \mathbb{C}\}$.

Service Definition

1. Functionality: A *ConfiguredService* provides a single function. This functionality is defined to include the function *signature*, *result* value, *preconditions* and *postconditions*.

Definition 1 *A service function is a 4-tuple* $f = \langle g, i, pr, po \rangle$*, where* g *is the function signature,* i *is the function result,* pr *is the precondition, and* po *is the postcondition. A signature is a 3-tuple* $g = \langle n, d, u \rangle$*, where* $n : string$ *is the function identification name,* $d = \{x | x \in \Lambda\}$ *is the set of function parameters and* $u : string$ *is the function address, the physical address on a network that can be used to call a function. For example, it can be an IP address. The result is defined as* $i = \langle m, q \rangle$*, where* $m : string$ *is the result identification name and* $q = \{x | x \in \Lambda\}$ *is the set of parameters resulting from executing the* ConfiguredService. *The precondition* pr *and postcondition* po *are data constraints. That is,* $pr :: z, z \subseteq \Lambda$ *and* $po :: z, z \subseteq \Lambda$

2. Nonfunctional properties: Typical nonfunctional properties associated with the service are pricing, shipping, and maintenance information. Pricing can be formalized as follows.

Definition 2 *Nonfunctional property list is* $\kappa = \langle p, \ldots \rangle$*, where* p *is the service cost and* $\ldots$ *denote other nonfunctional properties. The service cost* p *is defined as a 3-tuple* $p = \langle a, cu, un \rangle$*, where* $a : \mathbb{N}$ *is the price amount defined as a natural number,* $cu : cType$ *is currency tied to a currency type* $cType$*, and* $un : uType$ *is the unit for which pricing is valid. As an example,* $p = (100, \$, hour)$ *denotes the pricing of* 100\$*/hour. Other nonfunctional properties can be similarly defined using appropriate data types and included in* κ*.*

3. Attributes: These include some semantic information that is unique to a service.

Definition 3 *An attribute has a name and type, and is used to define some semantic information associated with the service. As an example, each* ConfiguredService *can be given a unique identifier, a version number, and type of release. They are defined as service attributes. The set of attributes is* $\alpha = \{(Dt, v_\alpha) | Dt \in \mathbb{T}, v_\alpha : Dt\}$.

Putting these three definitions together we arrive at the formal definition of a service given below.

Definition 4 *A service is a 3-tuple* $\sigma = \langle f, \kappa, \alpha \rangle$*, where* f *is the service function,* κ *is the set of nonfunctional properties, and* α *is the set of service attributes.*

The contract will include the following elements:

1. Trustworthiness: Trustworthiness properties are divided into two parts *ServiceTrust* which is related to *service provision*, and *ProviderTrust* which is related to the *service provider*.

Definition 5 ServiceTrust *is defined as the 4-tuple* $tr_{cs} = \langle \rho, \epsilon, \psi, \eta \rangle$*, where* ρ *is the safety guarantee,* ϵ *is the security guarantee,* η *is the availability guarantee, and* ψ *is the reliability guarantee. The safety guarantee includes time guarantee* ρ_t *and data guarantee* ρ_d*. We assume that* $time$ *is a generic type. The time guarantee is defined as* $\rho_t : time$*, the time the service takes to provide its function. The time guarantee will be turned into a predicate of the form* $t_e \leq t_b + \rho_t$ *where* t_b *and* t_e *refer to the beginning and ending times of service delivery. The data guarantee refers to the guarantee of satisfying data constraints of data, and is defined as* $\rho_d :: z, z \subseteq \Lambda$*. Let* H *denote the set of security protocols that the service provider has followed to guarantee confidentiality and integrity constraints. Then the set* $\epsilon = \{x | x \in H\}$ *defines the extent of security binding the service. Following the nature of security investigated in [MA11] we focus on* service security *and* data security. *Service security states that (1) for every service a request can be accepted only from a user who has permission to request that service, and (2) every service delivery will happen only if the service requester has permission to receive it. Data security states that (1) for every data parameter in a service request the service requester should have permission to* access *that data parameter, and (2) for every data parameter associated with the service delivery, the user receiving the service should have permission to* read *and* use *that data parameter. If a client does not have permission to send a request then the request will be ignored. Also, if a user does not have permission to receive a service, the service will not be sent. In [MA11] these security policies are implemented for components that provide services. By including* *the set $\epsilon = \{x | x \in H\}$ in a* ConfiguredService *description we are stating the mechanisms used to implement these security features. In addition, we may also add references to the security technology used in implementing the security policies and third party security firms using such technology. Such statements are a mean to convince the service requester on the strength of security attached to the service. The reliability guarantee refers to the guaranteed maximum number of failures in a unit of time, and is defined as $\psi : double$. It can be turned into a predicate $fre/t < \psi$, where fre is the total number of failures that occur during the total service execution time t, as defined in [MA11]. The availability guarantee refers to the guaranteed maximum time for repairs, and is defined as $\eta : time$. This can be turned into a predicate $t_{fi} - t_{si} < \eta$, where t_{si} is the time of failure i and t_{fi} is the time in which failure i was repaired, as defined in [MA11].*

Definition 6 ProviderTrust *is defined as a 3-tuple $tr_p = \langle ce, pg, re \rangle$, where ce is recommendations from other clients, pg is lowest prices guarantees and re is recommendations from independent organizations. Lowest price guarantee is represented by a flag $pg = (a | a : Boolean)$. It is a Boolean that is true when a* ConfiguredService *can guarantee its price to be lower than the price of any other* ConfiguredService *providing the same functionality. Client recommendations and recommendations from independent organizations can be defined as sets of ordered pairs. In $ce = \{(x, y) | x : CLIENT, y \in \{Low, BelowAverage, Average, AboveAverage, High\}\}$, the pair (x, y) represents a client x whose recommendation of the service is y. Likewise, in $re = \{(x, y) | x : ORGAN-IZATION, y \in \{Low, BelowAverage, Average, AboveAverage, High\}\}$, the pair (x, y) represents the recommendation y of an organization x.*

Formally, trustworthiness is defined using the above two definitions as:

Definition 7 *A trustworthiness property of a* ConfiguredService *is a composite property, written as a 2-tuple $\delta = \langle tr_{cs}, tr_p \rangle$, where tr_{cs},* ServiceTrust *and tr_p, the* ProviderTrust *are defined as explained above.*

We remark that not all components of δ may be relevant for a service, as shown in many later examples. In general, the trust domain, in which ce and pg are defined, must be a *complete lattice* [WA08b]. This property is essential in order to compare trust values of groups and compute minimum (maximum) among trust values. For the sake of simplicity, we assume in further discussion that trust values assumed by ce and re are whole numbers in the range $1 \ldots 5$, where 1 denotes Low and 5 denotes $High$. This assumption will enable us to calculate simple averages, maximum, and minimum of a set of trust values.

2. Legal issues: As part of the contract in a *ConfiguredService*, a set of legal rules that constrain the contract may be included.

Definition 8 *A legal issue is a rule, expressed as a logical expression in $\mathbb{C}$. A rule may imply another, however no two rules can conflict. We write $l = \{y | y \in \mathbb{C}\}$ to represent the set of legal rules.*

As an example, the legal rule 'If delivery takes more than 7 days from order, the shipping is free' can be formalized as $(delivery\_day > order\_day{+}7) \Rightarrow (shipping\_charge = 0)$. Similar approach can be used for all legal rules.

3. Context: Both *context information* and *context rules* are formally specified in a contract. These two parts provide context-awareness ability to *ConfiguredServices*.

Definition 9 *A context is formalized as a 2-tuple β = $\langle r, c\rangle$, where $r \in \mathbb{C}$, built over the contextual information c. Context information is formalized using the notation in [Wan06]: Let τ : $DIM \rightarrow I$, where DIM = $\{X_1,\ X_2,...,X_n\}$ is a finite set of dimensions and $I = \{a_1, a_2, ..., a_n\}$ is a set of types. The function τ associates a dimension to a type. Let τ(X_i) = a_i, $a_i \in I$. We write c as an aggregation of ordered pairs (X_j, v_j), where $X_j \in DIM$, and $v_j \in \tau$(X_j).*

Formally a contract is defined below.

Definition 10 *A contract is a 3-tuple μ = $\langle \delta, l, \beta\rangle$, where the service trustworthiness properties δ, the set of legal rules l and the context β are defined as presented above.*

Putting these definitions together we arrive at a formal definition for *ConfiguredService*.

Definition 11 *A* ConfiguredService *is a 2-tuple s = $\langle \mu, \sigma\rangle$, where μ is a contract, and σ is a service.*

4.3 Analysis

The contract part of a *ConfiguredService* has a legal binding between the service provider selling that service and the service requester buying that service. Consequently all the claims made by the service provider and all rules that will apply during and after service execution must *provably* remain true. In the computing literature the term *verification* is used to refer to proving correctness of program with respect to its specification (or a stated property), and the term *validation* is used to refer to proving the satisfaction of a program with respect to its requirements. We extend these concepts to analyze *ConfiguredServices*.

ConfiguredServices will be published by service providers, discovered by service requesters and executed on behalf of both. These are sequential actions, although the same sequence may be repeated cyclically. At each stage a *ConfiguredService* should retain the full spirit of legality, although the information content of the *ConfiguredService* might undergo changes. This implies that a proof of the claims made at a certain stage should not invalidate the proof of claims made earlier in the sequence. The analysis will be performed by different parties at each stage. The three main parties responsible for the analysis are the service provider, the service requester and an external trusted authority.

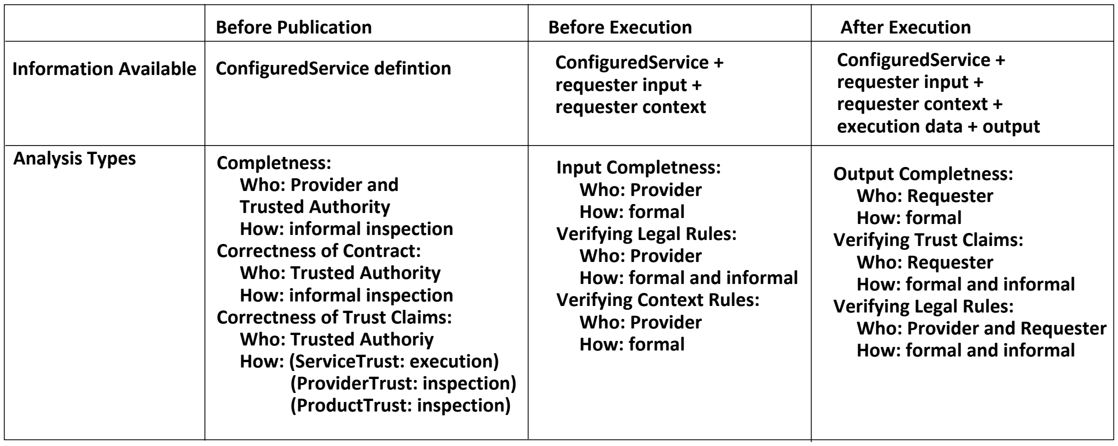
The analysis can be either 'formal' or 'informal'. Certain information content can be validated only by inspection or seeking recommendations, although contract fulfillment with respect to legal and business rule applications should be formally checked. We use the term validation to refer to 'informal' and 'semi-formal' analysis steps, and usually reserve the term verification to refer to formal analysis, such as verification using predicate logic resolution and model checking. The formal representation of *ConfiguredService* is mainly intended to enable a rigorous verification of it whenever possible.

The analysis is performed at three main stages, namely at those stages where information is incrementally added to a *ConfiguredService* description. The first analysis stage is before service publication. Both the trusted authority and the service provider cooperate in performing the analysis at this stage. The goal of analysis at this stage is to ensure that the

ConfiguredService description is as complete and correct as possible. In particular, the analysis will ensure that the service provider is not betraying or cheating the customers through false claims. Only those *ConfiguredServices* that pass this evaluation should be published. The second analysis stage is before service execution. At this stage the published *ConfiguredService*, and the input and the contextual information from a service requester are available. The analysis at this stage is performed by the service provider. The goal of this analysis is to ensure input completeness from the service requester and to ensure that legal rules pertaining to the sale are satisfied. The third analysis stage is after service execution. Most of this analysis will be done after service delivery, by both the service provider and the service receiver. Since the service receiver does not get access to the formal *ConfiguredService* description, the verification done by the service receiver can only be informal. However, the service receiver (requester) might delegate its responsibility to a third party trusted authority, in which case this trusted authority might be able to access the formal *ConfiguredService* description and the service source (from the service provider) and conduct a formal analysis. At this stage, the execution data and results are available in addition to all information from the previous two stages. The goal of analysis at this stage is to ensure that the service delivered to the service requester is indeed the service bought by the service requester, and all contractual obligations are met by both parties. Table 4 shows a summary of the analysis performed at each stage, persons responsible for performing the analysis and the type of analysis performed. In the rest of this section, each analysis stage is discussed in detail.

4.3.1 Analysis before Service Publication

A service provider defines a *ConfiguredService* and publishes it. Before publication the content of the *ConfiguredService* should be analyzed. The analysis might be performed by the service provider for quality control. A more critical analysis is performed by the trusted authority. The trusted authority should agree with the claims included in the *ConfiguredService* definition before allowing to publish the *ConfiguredService*. At this stage the informal and formal descriptions of a *ConfiguredService* are available.



	Before Publication	Before Execution	After Execution
Information Available	ConfiguredService defintion	ConfiguredService + requester input + requester context	ConfiguredService + requester input + requester context + execution data + output
Analysis Types	Completness: Who: Provider and Trusted Authority How: informal inspection Correctness of Contract: Who: Trusted Authority How: informal inspection Correctness of Trust Claims: Who: Trusted Authoriy How: (ServiceTrust: execution) (ProviderTrust: inspection) (ProductTrust: inspection)	Input Completness: Who: Provider How: formal Verifying Legal Rules: Who: Provider How: formal and informal Verifying Context Rules: Who: Provider How: formal	Output Completness: Who: Requester How: formal Verifying Trust Claims: Who: Requester How: formal and informal Verifying Legal Rules: Who: Provider and Requester How: formal and informal

Table 4: Analysis Stages

We propose that the analysis before publication focuses on the three essential properties *completeness*, *correctness*, and *verifiability*. Completeness refers to functional completeness in *service* definition, correctness refers to *contract* consistency and verifiability refers to the ability to verify the trust claims.

- **Completeness** In general, assuring completeness of information is hard. We are insisting only on functional completeness, in the sense that the *preconditions* and *postconditions* are sufficient to uniquely define the service functionality, and the nonfunctional properties are sufficient to describe the quantifiable attributes of significance to the service. In case of incompleteness, the service provider should add more nonfunctional properties to convince the consumer about the service functionality. Incompleteness is not an error, and can be remedied. It is also essential to ensure that the information defined in the *attributes* section is sufficient to identify the service types, the service providers, and the product type (if applicable). As an example, displaying a Car (image from the inventory) that satisfies the *Product Type* specification is an acceptable sufficient proof to convince the buyer that a car that meets its specification exists. Analyzing completeness is a manual process (or user-assisted semi-automated process) in which the service provider or the trusted authority to whom the task is delegated, leads the validation steps. In the Car Rental (Table 3) and the Buy/Ship (Table 1/ Table 2) Book examples, functional completeness is to be achieved by such inspection.

- **Correctness of Contract Rules** The contract rules will include the legal rules and context rules. The analysis is static in the sense that no rule can be fired before contract execution. The static analysis may be performed both by inspection and by formal resolution principles of propositional logic. The legal rules will be checked to make sure they are neither ambiguous nor contradictory. Contradictions can be formally checked, however semantic ambiguity is hard to check formally. For example, the two rules "r_1: no discount on weekdays", and "r_2: '15$\%$ discount for students" cannot be applied simultaneously in a 'week day' context. So, rule r_2 must be restated as "15$\%$ discount for students on weekends". Such validations are performed by the trusted authority.

  The context information will be validated by the trusted authority to ensure its correctness and its expressivity in conveying the service provider's contextual information. As an example, if the provider claims that the business is located in Canada the trusted authority should be able to manually verify this claim. The context rules cannot be evaluated at this stage. However they should be validated to make sure they contain no contradictions. This is also done by the trusted authority. As an example, assume that the service provider location is given as $Montreal$ city. Suppose one context rule states that "the service requester should be located within 100 Kms from service provider location". This rule can be validated only at service execution time. If there exists a second context rule stating "the service may be provided anywhere in Canada", then obviously these two rules are not consistent. The trusted authority can catch such contradictory context rules at this stage.

- **Validating Trust Claims** The trusted authority has sufficient knowledge and skills to verify and validate the correctness of service trust claims. The main goal of such verification and validation is to ensure that the service provider is not stating misleading information or trying to cheat consumers on the quality of service. In the *ProviderTrust* part, recommendations from independent organizations and clients are included. Such properties can only be manually verified. The trusted authority will contact the independent organizations to validate the correctness of such recommendations. It will review the history of use and contact clients to ensure the correctness of the claimed client recommendations. To validate the correctness of the claims in *ServiceTrust* part the trusted authority might use a variety of techniques:

- If a claim is related to a product, the trusted authority might request the service provider to 'show' the product. The existence of the product that meets its specification is a sufficient proof of the claim.
- If a claim is related to a service associated with a product the trusted authority will get the product, and the formal description of its services. The service claim, regarded as behavior, is then resolved against the service specification. Executing the service specification, the trusted authority can validate the claim.
- If a claim is related to a service the trusted authority will get access to the service implementation, use the interfaces and tools that are given to it by the service provider, and execute the service. The claims, regarded as properties stated in the formal *ConfiguredService* description, are then formally verified in service execution. This process might include formal verification using formal verification tools, or visualization or animation to view and inspect the service behavior. In fulfilling its task, we are expecting the trusted authority to be as powerful as the *Trusted Computer System Evaluation Standard (Orange Book)* [Boo] in its authority to demand information from service providers, its ability to formally assess the submitted implementations against the formalized claims of *ConfiguredServices*, and approving the publication of only those *ConfiguredServices* which pass its assessment.

4.3.2 Analysis before Service Execution

To execute a *ConfiguredService*, the service provider receives a service request associated with required inputs. The inputs will contain input parameters and the contextual information of the service requester. But before executing the *ConfiguredService*, the service provider needs to verify and validate the inputs with respect to the *ConfiguredService*.

We propose that the analysis before execution focuses on the three essential properties *input completeness*, *legal rules analysis*, and *context rules verification*.

- **Verifying Input Completeness** The input parameters received from the service requester should be verified to be complete. Input is complete if the following two conditions are satisfied (1) all required input parameters are provided by the service requester, and (2) the input is sufficient to satisfy the precondition of service functionality. To verify the first condition, the service provider will verify that the statement $((input_1 \neq null) \wedge (input_2 \neq null) \wedge \ldots \wedge (input_n \neq null))$ is equal to true. To verify the second condition, the service provider will verify that the precondition evaluates to true. This is done by substituting the variables in the preconditions with the actual values. As an example the precondition $valid(CreditCard) \wedge valid(DrivingLicense)$ from Table 3 is evaluated by substituting the parameters 'CreditCard' and 'DrivingLicense' with their actual values, and invoking predefined functions.

- **Verifying Legal Claims** After receiving the input from the service requester, the service provider needs to ensure (1) input data received will not contradict a legal rule, and (2) the input received is sufficient to evaluate all the rules that should be satisfied in order to sell the service. Verifying legal rules can either be formal and or informal. As an example, assume that a service requester wants to buy 10 books. But in the legal section of the contract, there is a rule that states “maximum of 5 books may be bought by a customer”. In this case, the input contradicts the legal rule. This analysis can be performed formally by the service provider. As a second example, let there be a rule stating “only credit card payments are allowed”. But, the service requester does not provide his credit card information. This legal rule cannot be evaluated. The analysis should ensure that the service is executed only after all input that can validate the legal rules are received from the service requester.

- **Evaluating Context Rules:** The service provider should formally validate that the contextual information supplied by the service provider satisfies the context rules. The contexts defined in the *Context Info* part of Table 3 use the three dimensions LOC (location information), $DATE$ (the day the car is rented) and $TIME$ (the time of checkout). Each dimension is associated with a value. Thus, taken as a whole we get a multidimensional perspective of the car rental context. As an example, $c = [LOC : NewYork, DATE : 09/09/2011, TIME : 9]$ is a specific instance of the car rental context. A legal rule might require a contextual evaluation. An example of such a rule is "only those who have an American driver license can rent a car". So, the usual validity check done as part of 'input completeness' must be extended to include the context dependent legal rules. We explain in Section 4.3.3 how context dependent evaluation is formalized.

4.3.3 Analysis after Service Delivery

The information available at this stage includes the *ConfiguredService* definition, the inputs from the requester including context, and the execution data and result. The execution data includes the statistics of the execution process itself. An example of execution data is, "during the execution the service failed and the recovery took 5 minutes". The execution result includes any partial output from service execution or service termination.

The contract of the *ConfiguredService* contains the rights and responsibilities of service requesters and the service provider. Hence, it is essential to verify and validate that each party has satisfied his obligations during service execution. Not satisfying an obligation might result in penalties and legal consequences as stated in the contract. The verification and validation is both performed by the service requester and provider.

We propose that the three essential properties for analysis are *output completeness*, fulfillment of *trustworthiness properties* and satisfaction of *context-dependent legal rules*. Below is a detailed discussion of the analysis for each of these properties.

- **Output completeness** The output parameters received from the service provider should be verified to be complete by the service requester. Output is complete if the all required output parameters are provided by the service provider, and the output is sufficient to satisfy the postconditions of service functionality. If n output parameters are expected by the service requester then the statement $((output_1 \neq null) \wedge (output_2 \neq null) \wedge \ldots \wedge (output_n \neq null))$ must evaluate to true. To verify the satisfaction of the postcondition, the service requester will have to evaluate the postcondition and verify that it is true. In case the verification is unsuccessful the service requester should invoke the legal rules in the contract to remedy the situation. If the verification is successful, the service provider might invoke the rules related to consumer obligation. Therefore, the trusted authority must be part of this loop to verify output completion.
- **Trustworthiness properties** The service requester will have to check that the stated claims are fulfilled. Because the service requester received a certificate of trust from the trusted authority when buying the service and the published *ConfiguredService* is only informal, it is only appropriate that the verification at this stage is only informally done, may be by inspection by the service requester. Below are examples of trustworthiness claims and how they can be verified.
    - Safety (time): The service requester monitors the service execution period and checks that it satisfies the safety claim stated in the *ConfiguredService*.
    - Safety (data): The service requester monitors the service execution to ensure the data constraints stated in the *ConfiguredService* are not violated.
    - Security: The service provider might guarantee the use of some security

        policies, such as encryption, in service delivery. After service execution, the service requester can evaluate manually the satisfaction of such claims, because a decryption method would be sent to him by the service provider. If the service provision violated privacy guarantees then the service receiver should invoke the contract rules for seeking recourse.

    - Reliability: Reliability can be verified by the service receiver, say by counting the number of failures during service delivery and validating the reliability claim.

    - Availability: If service is not available after service delivery, then the service receiver observes the restart time and ensure that the time between failure and restart is less than the stated availability claim the *Configured-Service*.

- **Legal rules** The legal rules that govern the rights and responsibilities of the service provider and service requesters after service delivery are to be evaluated at this stage. Such evaluations will reveal the consequences of not satisfying the contract rules. As an example, consider the *collision and liability insurance* rule in Table 3. Formally the rule is

  $$NotCovered(COLLISION \wedge LIABILITY) \wedge accident$$
  $$\Rightarrow Charge(CreditCard(cardnumber), \$30,000),$$

  where $NotCovered$, $Charge$, and $CreditCard$ are predicate names, $COLLISSION$, $LIABILITY$, are literals (meaning the same as in the insurance semantic domain), $accident$ is a proposition, and $cardnumber$ is a variable. This rule is fired only during service execution, namely when an accident happens during the use of the rented car. This rule will automatically modify the contract rule on liability. Verifying that this rule was indeed enforced is the responsibility of the service provider. Service delivery contexts and other contexts that arise during or after service execution will require context-dependent legal rules to be verified. We consider the example in Table 3. When the car is rented we do not know the exact context in which the car will be returned. However we can model the acceptable car return *situation* as a logical expression involving the dimension names in the rental context, and including that as a rule in the *ConfiguredService*, as shown in Table 3. The acceptable car return situation corresponding to context c is $(LOC = retloc) \wedge (DATE{+}7 \ \geq \ returndate)$. Assume the car is returned in $Boston$ on September 14, 2011. The situation is evaluated by replacing the dimension LOC by $NewYork$, the dimension $DATE$ by $09/09/2011$, the variable $retloc$ by $Boston$, and the variable $returndate$ by $09/14/2011$. It is easy to see that the situation predicate is false. So, the conclusion is that contract is violated. In summary, the legal rules that are context-dependent can be automatically verified at the stated contexts by following these steps:

1. Formulate the legal rules as predicates involving the $DIM \ ENSION$ names and variables.
2. To evaluate a predicate p in a context c,
   - $*$ replace each $DIMENSION$ name in p by the tag value if it is a $DIMENSION$ in c,
   - $*$ substitute the variables in p by their respective values, as provided either in the *ConfiguredService* or in the environment of evaluation,
   - $*$ if p still has $DIMENSION$ names or variables then p cannot be evaluated in the context c; otherwise p is now a proposition which evaluates to either true or false.

The evaluation process being simple, it can be made part of a service provider system software or part of a trusted authority system software.

After the execution of the service and the analysis of the service requester, the service requester can create a trust level for the service according to its experience. This information, if sent to the service provider, will help the service provider in the future announcements of service claims. Also the service requester can add a recommendation to the service provider. This recommendation will either increase or decrease the service provider rating.

4.4 Flexible Contracts

If the contract part in a *ConfiguredService* is strict, in the sense that it is not allowed to change, then it forces the service requester to either take it or leave it. In order to increase economic value and improve trusted transactions it is necessary that the contract part in a *ConfiguredService* remains 'flexible'. The flexibility goal is to allow consistent contract modifications, some by negotiation and some by automatic triggering of contract rules, whenever necessary. A flexible contract is still 'strict' in the sense that it will remain precise after changes for a formal interpretation and enforcement. In this section we study the different types of contract modifications and their causes, and offer syntactic simplification for drafting flexible contracts.

The service part in a *ConfiguredService* is not allowed to change. If a change is demanded on the service functionality in a *ConfiguredService*, the service provider might have to create a new *ConfiguredService* in which an entirely new contract part will bind the new service. A *ConfiguredService* which has a flexible contract is 'self-evolving', because it generates a set of *ConfiguredServices* such that all of them have identical service part while contracts are different. We achieve syntactic simplification of a self-evolving *ConfiguredService*, because we need to create only one template. We achieve semantic generality by generating many *ConfiguredServices* when the contract parts are modified through a simple syntactic extension, as discussed in Section 4.4.2.

4.4.1 Types and Causes of Contract Modifications

The following reasons are compelling to allow different types of flexible contracts in a service-oriented system.

- *Changes triggered by service provider:* Business rules might change. Consequently changes to nonfunctional and trustworthiness properties might become necessary. Such changes should be only towards improving service quality. For example, price increases should not affect consumers who had already signed contracts. In this case, the legal statement "*Price changes before service execution will not affect* *the signed contract*" might be added to the contract part of the *ConfiguredService*. Context information might change. Such a change should not affect those who have already signed contracts for receiving services. The context change might invalidate some of the rules in the contract, because some conformance conditions might fail. So, the service provider must modify the contract to suit the new context and assure the customers that such a change will not affect the trust attributes, and service delivery constraints. This will enhance the consumer trust. The statement "*In case the service provider context changes the consumers will be informed and existing contract will be respected by the service provider*" can be added to the legal rules. New version of older services might be introduced by a service provider. The contract part in a new version must be an incremental extension of the old contract, otherwise old contracts might be violated. At the time of service delivery, the service provider may have a new *ConfiguredService* in which the functionality is the same as the old one, however nonfunctional properties and trust properties have been improved. For example, the price may be less and it is more secure. To earn consumer trust, it is advisable for the service provider to inform the consumer on this alternative service, and with consumer's consent modify the old contract. To deal with such situation, the statement "*If the quality attributes of the functionality are improved the consumers may freely select the new improved ConfiguredService*" can be added to the legal rules. Exceptions that arise during service delivery should be addressed by automatic changes to the legal rules of the current contract. It is wiser to automate this process through the introduction of rules that trigger changes to other rules.

- *Changes triggered by service requester:* The service requester might demand some changes in the contract part, for instance privacy in service delivery. This in turn might initiate a negotiation process between the service requester and service provider. In general, negotiation is an integral part of service-centric activity. Typically negotiation starts only after a *ConfiguredService* is selected by a service requester, and the negotiation process usually centers only on the contract part of the selected service.

A negotiable contract might mention a latency period between the signing of the contract and its execution, thus allowing the retailer and consumer to enrich or modify or cancel the contract during this period. This kind of flexibility is most common in business contracts. After service selection but before service delivery it is likely that the service delivery context changes. This in turn might invalidate many legal rules in the contract part of the selected *ConfiguredService*, unless they are amended. To prevent this and sustain consumer trust, the service provider and the service requester might renegotiate or the service provider is given the option to access and get a new *ConfiguredService* that satisfies the new context situation. To deal with context change, the statement "*In case the delivery context changes, the consumer may select a new ConfiguredService without penalty.*" can be added to the legal rules. It is wiser to automatically modify existing rules through the introduction of rules that trigger changes to other rules that exist in an already agreed upon contract.

- *Service failures:* A service might fail during service delivery, may be because of communication failure or the service itself has become unavailable. If service stoppage violates the contract, the consumer must be compensated. If the service is no more available, then the consumer must be allowed to search the service registry for another *ConfiguredService* in which the functionality is *equivalent* to the functionality of the previously chosen service. In the later case, some price concession must be given. To deal with service failures, the statements "*The consumer will be compensated for service interruption. The consumer may choose new* ConfiguredService*, with a 10% discount in price, in case the chosen service becomes unavailable, or the consumer may elect to cancel the contract with no penalty*" can be added to the legal rules. Regardless of the incentive, restarting the entire process from discovering new service and following the service processing flow until its delivery is costly and time consuming activity. To minimize the damage established to trust we have considered in Section 7.6 a method that discovers alternating services. Essentially, when a service fails another service from the 'ranked set of equivalent services' is automatically chosen and its contract is negotiated before continuing with service delivery.

```
ConfiguredService S
    includes ConfiguredService S_1
    extended-by {
        ⋮
    }
    modified-as{
        ⋮
    }
```

Figure 6: *ConfiguredService* Extension Syntax

4.4.2 Syntactic Issues - Extension through Inclusion and Modification

In this section two constructs are discussed for syntactically modifying the contract part in a *ConfiguredService*. One construct is the **extended-by** clause and the other is the **modified-as** clause.

To enrich a *ConfiguredService* the **extended-by** clause is to be used. An enrichment adds more attributes (information) to the sections in it, with no changes to existing content. This allows incremental addition to consumer data, constraints, nonfunctional properties, trustworthiness properties, legal and exception rules, and context part. The addition of information may include new context information which does not *invalidate* existing *contract* and *context* sections.

To modify the information and rules in a *ConfiguredService* the **modified-as** clause is to be used. No new information may be included within **modified-as** clause. Trustworthiness properties, legal and exception rules, and context may be changed by necessity. The syntax that handles both types of extensions is given in Figure 6.

The syntax in Figure 6 is a shorthand for creating self-evolving *ConfiguredServices*. In the **includes** clause one *ConfiguredService* name S_1 is listed. This *ConfiguredService* will be extended as stipulated by the **extended-by** and **modified-as** clauses. If the **extended-by** clause is absent, the effect is only to modify the included *ConfiguredService*. If the **modified-as** clause is absent, the effect is only to enrich the included *ConfiguredService*. Table 5 shows a modification of the *Car Rental ConfiguredService* in its first column, and

Modified Car Rental	Extended Car Rental
ConfiguredService MCar-Rental **includes** *ConfiguredService* Car Rental **modified-as** { **Contract:** Legal Rules: *car return:* no fee when returned to the location where rented 50$ additional fee if returned to another location *collision and liability insurance:* fully covered }	*ConfiguredService ECar-Rental* **includes** *ConfiguredService* Car Rental **extended-by** { **Service:** Attributes: *Consumer Data:* Second Driver: driver license:ALP10091878 age: 52 **Contract:** Trust Attributes: trust recommendation: AAA (5 star) , Consumer (9/10) }

Table 5: A Modified and Extended ConfiguredService

an enrichment of the *Car Rental ConfiguredService* in its second column.

4.4.3 Semantic Issues

The information listed within the **extended-by** clause is added to the information in the included *ConfiguredService* in such a way that the result is a *ConfiguredService* template. This is achieved by 'pairwise' conjoining of the information contents in the included *ConfiguredService* and the **extended-by** clause. Rules from the **extended-by** clause that conflict with the included *ConfiguredService* will be removed. The resulting *ConfiguredService* is subject to further analysis, as explained in Section 4.3.

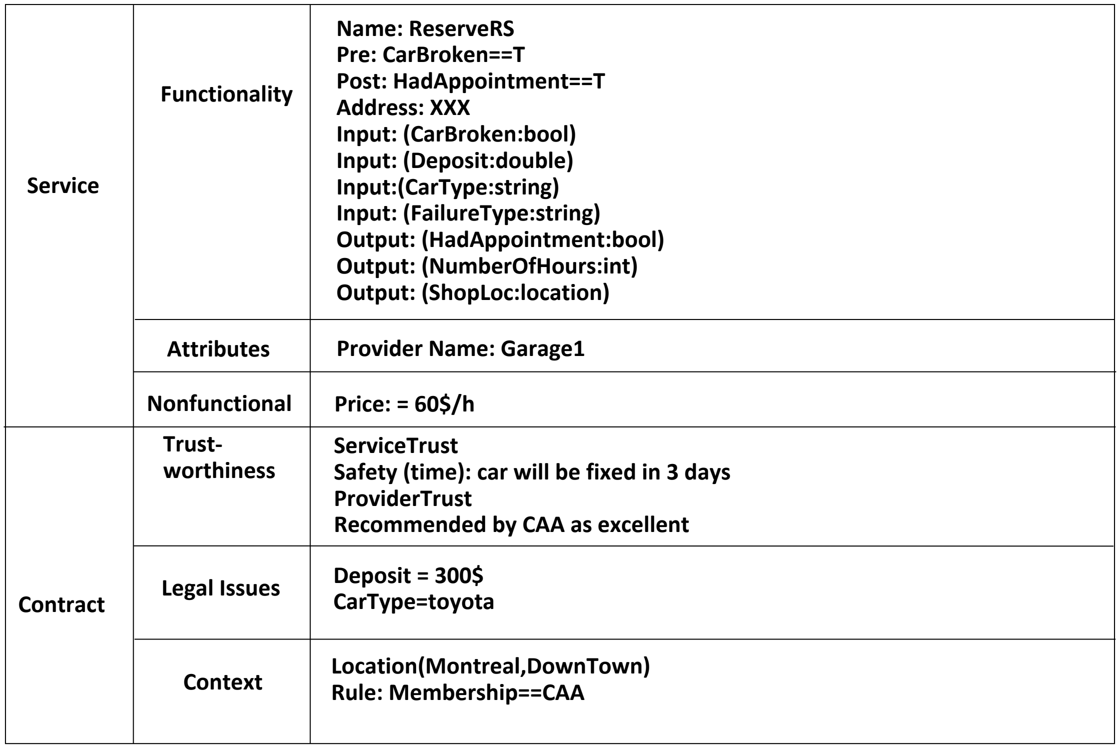
The information listed within the **modified-as** clause will 'overwrite' the information in the included *ConfiguredService* in such a way that the result is a *ConfiguredService* template. To achieve this it is necessary that every rule type in the *modified-as* clause is a rule type in the original *ConfiguredService*. No new rule type is allowed in the **modified-as** clause. Assuming that the included *ConfiguredService* had information consistency and a set of conflict-free rules we expect the resulting *ConfiguredService* to have these properties. That is, if an inconsistency arises due to overwriting, that rule type is removed from the **modified-as** clause. The resulting *ConfiguredService* is subject to further analysis, as explained in Section 4.3.

Modifications to contract do not change the functionality of the included *ConfiguredService*. A flexible contract can modify itself if a rule triggers changes to other rules either during or after contract execution. As an example, assume that the *ConfiguredService MCar-Rental* in Table 5 is modified again by the introduction of the new rule “Any person with a valid driver license and who is not included in the service contract may drive the rented car, however the collision and liability insurance contract is null and void”. This rule takes effect only when the rented car is driven by a person satisfying the constraints stated in the rule. Such a situation may never arise. Yet, the net effect is the automatic production of a new contract in which “the collision and liability insurance” rule of the original contract is deleted and the new rule is included.

4.5 Case Study - Auto Roadside Emergency Service

In this section, we introduce a case study chosen from the *automotive industry*. This case study has been used in the literature of SOA by several researchers [tBGKM08], [Koc07] and [BK07]. This example will be used throughout this thesis. We first give the background to ‘product functionality’, which in this example is a car, and the ‘behavior of the product’ which in this case is the interaction between engine, sensors, and actuators. Services are related to a specific behavior of the car.

Vehicles today are equipped with multiple sensors and actuators that provide the driver with services that assist in driving the vehicle more safely, such as vehicle stabilization systems. Here we will focus on the *road assistance scenario*. This scenario deals with the case of a car failure. For example, the oil lamp in the car might turn red to indicate a low oil level. This will trigger the diagnostic system to analyze the values obtained by the oil level sensor. The diagnostic system then reports, for example, the failure in one cylinder head and the car is no longer drivable. This information and the location information obtained from the GPS system is sent to the road assistance center. The road assistance center will



Service	Functionality	Name: ReserveRS Pre: CarBroken==T Post: HadAppointment==T Address: XXX Input: (CarBroken:bool) Input: (Deposit:double) Input:(CarType:string) Input: (FailureType:string) Output: (HadAppointment:bool) Output: (NumberOfHours:int) Output: (ShopLoc:location)
	Attributes	Provider Name: Garage1
	Nonfunctional	Price: = 60$/h
Contract	Trust-worthiness	ServiceTrust Safety (time): car will be fixed in 3 days ProviderTrust Recommended by CAA as excellent

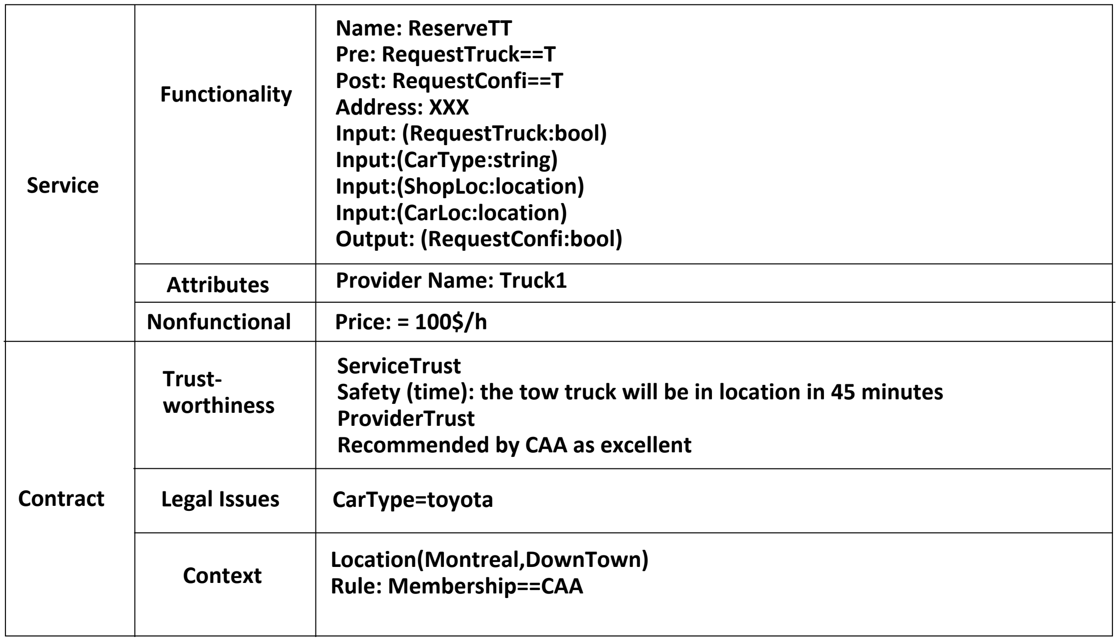
	Legal Issues	Deposit = 300$
		CarType=toyota

	Context	Location(Montreal,DownTown) Rule: Membership==CAA

Table 6: RepairShop ConfiguredService

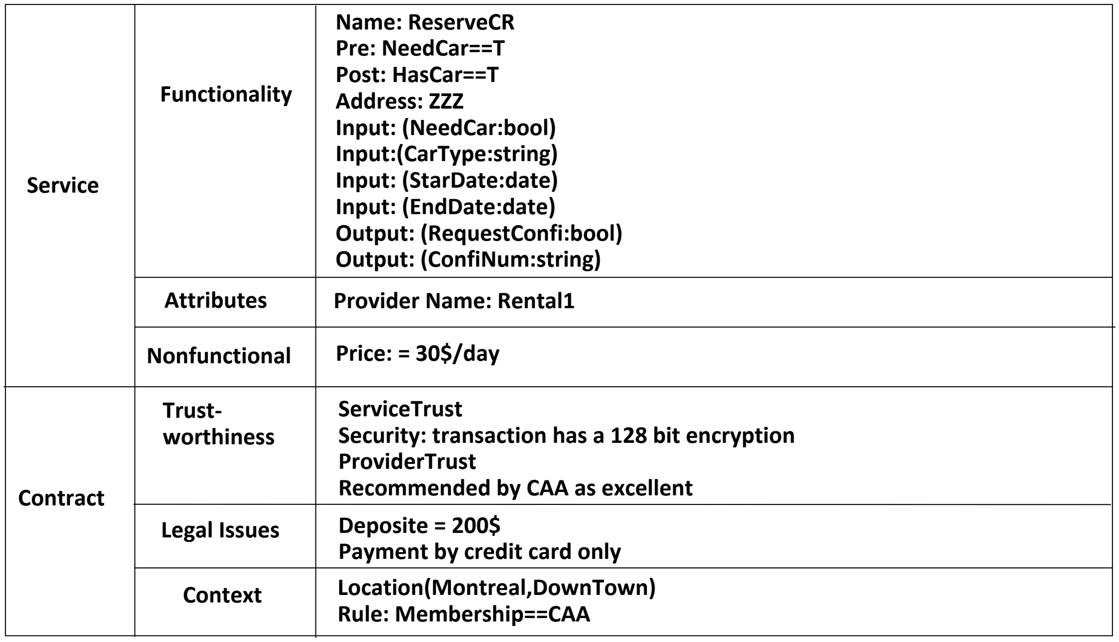
use this information to identify the appropriate *repair shop*, *tow truck* and *car rental* service providers and inform the driver. The driver will select a repair shop. The diagnostic results are then sent automatically to the selected repair shop. This will allow the repair shop to identify the spare parts needed to repair the car. After that, the driver orders a tow truck and a rental car. The GPS coordinates of the vehicle and repair shop are sent to the tow truck. The driver is required to deposit a security payment before being able to reserve a repair shop or a car rental. Each service can be denied or canceled, causing an appropriate compensation activity.

In this example, we identify three *ConfiguredServices*, whose detailed definitions are shown in Tables 6, 7 and 8.



Service	Functionality	Name: ReserveTT Pre: RequestTruck==T Post: RequestConfi==T Address: XXX Input: (RequestTruck:bool) Input:(CarType:string) Input:(ShopLoc:location) Input:(CarLoc:location) Output: (RequestConfi:bool)
	Attributes	Provider Name: Truck1
	Nonfunctional	Price: = 100$/h
Contract	Trust-worthiness	ServiceTrust Safety (time): the tow truck will be in location in 45 minutes ProviderTrust Recommended by CAA as excellent
	Legal Issues	CarType=toyota
	Context	Location(Montreal,DownTown) Rule: Membership==CAA

Table 7: TowTruck ConfiguredService



Service	Functionality	Name: ReserveCR Pre: NeedCar==T Post: HasCar==T Address: ZZZ Input: (NeedCar:bool) Input:(CarType:string) Input: (StarDate:date) Input: (EndDate:date) Output: (RequestConfi:bool) Output: (ConfiNum:string)
	Attributes	Provider Name: Rental1
	Nonfunctional	Price: = 30$/day
Contract	Trust-worthiness	ServiceTrust Security: transaction has a 128 bit encryption ProviderTrust Recommended by CAA as excellent
	Legal Issues	Deposite = 200$ Payment by credit card only
	Context	Location(Montreal,DownTown) Rule: Membership==CAA

Table 8: CarRental ConfiguredService

4.5.1 Formal Representation

RepairShop: Let rs denote the *ConfiguredService* for providing a Repair Shop who provides the services described in Table 6. The formal notation of *ConfiguredService* rs is s_{rs} = $\langle \mu_{rs}, \sigma_{rs} \rangle$, where the tuple components are explained below.

- Service: $\sigma_{rs} = \langle f_{rs}, \kappa_{rs}, \alpha_{rs} \rangle$ where,

  1. Function: $f_{rs} = \langle g_{rs}, i_{rs}, pr_{rs}, po_{rs} \rangle$ where,

     – Function signature: $g_{rs} = \langle n_{rs}, d_{rs}, u_{rs} \rangle$, where $n_{rs} = (ReserveRS)$ is the name, $d_{rs} = \{(CarBroken, bool), (deposit, double), (CarType, string), (failureType, string)\}$ are input data parameters, and $u_{rs} = (XXX)$ is the address.

     – Function result: $i_{rs} = \langle m_{rs}, q_{rs} \rangle$, where $m_{rs} = (ResultRS)$ is the name and the set of output data parameters is $q_{rs} = \{(HasAppointment, bool), (numberOfHours, int), (ShopLoc, location)\}$.

     – Function precondition: $pr_{rs} = (CarBroken == true)$.

     – Function postcondition $po_{rs} = (HasAppointm\ ent == true)$.

  2. Nonfunctional: $\kappa_{rs} = \langle p_{rs} \rangle$, $p_{rs} = \langle a_{rs}, cu_{rs}, un_{rs} \rangle$, where $a_{rs} = (60)$ is the cost, $cu_{rs} = (dollar)$ is the currency, and $un_{rs} = (hour)$ is the pricing unit.

  3. Attributes: $\alpha_{rs} = \{(name = Garage1)\}$.

- Contract: $\mu_{rs} = \langle \delta_{rs}, l_{rs}, \beta_{rs} \rangle$ where,

  1. Trustworthiness: $\delta_{rs} = \langle tr_{cs}, tr_p \rangle$ where,

     – ServiceTrust: $tr_{cs} = \langle \rho_t \rangle$ where $\rho_t = 3days$. This can be written in predicate logic as $time_{end} \leq time_{start} + 3$.

     – ProviderTrust: $tr_p = \langle re \rangle$ where $re = \{(CAA, Excellent)\}$.

  2. Legal: $l_{rs} = \{(deposit = 300), (CarType == toyota)\}$ where the deposit amount is 300 and the car type is toyota.

3. Context: β_{rs} = $\langle r_{rs}, c_{rs}\rangle$, where $r_{rs} = \{(membership == caa)\}$ is the context rule and $c_{rs} = \{(Location, (Montreal, Canada))\}$ is the contextual information of the repair shop service provider.

TowTruck: Let tt denote the *ConfiguredService* for providing a Tow Truck who provides the services described in Table 7. The formal notation of *ConfiguredService* tt is s_{tt} = $\langle \mu_{tt}, \sigma_{tt}\rangle$, where the tuple components are explained below.

- Service: $\sigma_{tt} = \langle f_{tt}, \kappa_{tt}, \alpha_{tt}\rangle$ where,

  1. Function: $f_{tt} = \langle g_{tt}, i_{tt}, pr_{tt}, po_{tt}\rangle$ where,

     - Function signature: $g_{tt} = \langle n_{tt}, d_{tt}, u_{tt}\rangle$, where $n_{tt} = (ReserveTT)$ is the name, $d_{tt} = \{(RequestTruck, bool), (CarType, string), (ShopLoc, location), (CarLoc, location)\}$ are input data parameters, and $u_{tt} = (YYY)$ is the address.
     - Function result: $i_{tt} = \langle m_{tt}, q_{tt}\rangle$, where $m_{tt} = (ResultTT)$ is the name and the set of output data parameters is $q_{tt} = \{(RequestConfi, bool)\}$.
     - Function precondition: $pr_{tt} = (RequestTruck == true)$.
     - Function postcondition $po_{tt} = (RequestConfi == true)$.

  2. Nonfunctional: $\kappa_{tt} = \langle p_{tt}\rangle$, $p_{tt} = \langle a_{tt}, cu_{tt}, un_{tt}\rangle$, where $a_{tt} = (100)$ is the cost, $cu_{tt} = (dollar)$ is the currency, and $un_{tt} = (hour)$ is the pricing unit.
  3. Attributes: $\alpha_{tt} = \{(name = Truck1)\}$.

- Contract: $\mu_{tt} = \langle \delta_{tt}, l_{tt}, \beta_{tt}\rangle$ where,

  1. Trustworthiness: $delta_{tt} = \langle tr_{cs}, tr_p$ where,

     - ServiceTrust: $tr_{cs} = \langle \rho_t\rangle$ where $\rho_t = 45 minutes$. This can be written in predicate logic as $time_{arrive} \leq time_{order} + 45$.
     - ProviderTrust: $tr_p = \langle re\rangle$ where $re = \{(CAA, Excellent)\}$.

  2. Legal: $l_{tt} = \{(method ==" Cash")\}$ where the payment method is cash only.

3. Context: β_{tt} = $\langle r_{tt}, c_{tt} \rangle$, where $r_{tt} = \{(membership == caa)\}$ is the context rule and $c_{tt} = \{(Location, (Montreal, Canada))\}$ is the contextual information of the tow truck service provider.

CarRental: Let cr denote the *ConfiguredService* for providing a Car Rental who provides the services described in Table 8. The formal notation of *ConfiguredService cr* is s_{cr} = $\langle \mu_{cr}, \sigma_{cr} \rangle$, where the tuple components are explained below.

- Service: $\sigma_{cr} = \langle f_{cr}, \kappa_{cr}, \alpha_{cr} \rangle$ where,

  1. Function: $f_{cr} = \langle g_{cr}, i_{cr}, pr_{cr}, po_{cr} \rangle$ where,

     - Function signature: $g_{cr} = \langle n_{cr}, d_{cr}, u_{cr} \rangle$, where $n_{cr} = (ReserveCR)$ is the name, $d_{cr} = \{(NeedCar, bool)$, $(CarSize, string)$, $(StarDate, date)$, $(EndDate, date)\}$ are input data parameters, and $u_{cr} = (ZZZ)$ is the address.
     - Function result: $i_{cr} = \langle m_{cr}, q_{cr} \rangle$, where $m_{cr} = (ResultCR)$ is the name and the set of output data parameters is $q_{cr} = \{(HasCar, bool), (ConfNum, string)\}$.
     - Function precondition: $pr_{cr} = (NeedCar == true)$.
     - Function postcondition $po_{cr} = (HasCar == true)$.

  2. Nonfunctional: $\kappa_{cr} = \langle p_{cr} \rangle$, $p_{cr} = \langle a_{cr}, cu_{cr}, un_{cr} \rangle$, where $a_{cr} = (30)$ is the cost, $cu_{cr} = (dollar)$ is the currency, and $un_{cr} = (day)$ is the pricing unit.
  3. Attributes: $\alpha_{cr} = \{(name = Rental1)\}$.

- Contract: $\mu_{cr} = \langle \delta_{cr}, l_{cr}, \beta_{cr} \rangle$ where,

  1. Trustworthiness: $delta_{cr} = \langle tr_{cs}, tr_p$ where,

     - ServiceTrust: $tr_{cs} = \langle \epsilon_{cr} \rangle$ where $\epsilon_{cr} = \{(encryption = 128)\}$.
     - ProviderTrust: $tr_p = \langle re \rangle$ where $re = \{(CAA, Excellent)\}$.

  2. Legal: $l_{tt} = \{(deposit = 200), (method ==" creditCard")\}$ where the deposit is 200$ and the payment method is credit card only.

3. Context: β_{cr} = $\langle r_{cr}, c_{cr}\rangle$, where $r_{cr} = \{(membership == caa)\}$ is the context rule and $c_{cr} = \{(Location, (Montreal, Canada))\}$ is the contextual information of the car rental service provider.

4.5.2 Analysis

We assume that each *ConfiguredService* has been analyzed before publication. In this section, we focus on the two types of analysis before execution and after execution. Below is a discussion of the analysis performed on the RepairShop *ConfiguredService*. Similar analysis is performed on other *ConfiguredServices*. The analysis performed on the RepairShop *ConfiguredService* can be summarized as follows:

- Before execution of service the following properties will be verified:
  - Input completeness: $((Deposit == depositammount) \wedge (credit\_card \neq null) \wedge (CarType \in Models) \wedge Acceptable(FailureType))$.
  - Context Information: $LOCATION = ((Montreal, downtown))$.
  - Legal claims: $(Deposit \geq 300)$, and $(CarType == toyota)$.
  - Context rules: $(membership(CarDriver) == CAA))$ will be formally verified.
- After execution the following properties will be verified.
  - Output completeness: $((HadAppoitment == true) \wedge (NumOfHours \neq null) \wedge (ShopLoc \neq null))$.
  - Trustworthiness properties: $(time_{end} \leq time_{start} + 72)$, assuming that time is measured in hours.

4.6 Summary

In this chapter, the novel *ConfiguredService* concept has been introduced, its constituent elements have been formally described. The essential properties of *ConfiguredService* have been enumerated and their incremental analysis discussed in three stages. The notion of flexible contracts has been introduced, and a formal syntax and semantics have been proposed for formally representing evolving *ConfiguredServices*. The chapter is concluded with an emergency roadside service example.

Chapter 5

Static Service Composition

Service-oriented applications are created by composing services together to create new services that provide more complex functionalities. Service composition may be attempted either at service publication time or at service execution time. The former is called *static* service composition and the later is called *dynamic* service composition. Static composition is done by the service provider driven by his business goals, usually driven by value added economic goals. Dynamic service composition, discussed in Chapter 8, is driven by user's demands at service provision contexts and is usually performed by the service provision framework. In this chapter, we focus on static service composition.

From the composition approaches discussed in Chapter 2 it is clear that most of the proposed approaches are not formal. There are a few exceptions, however these formal approaches focus only on composing service functionalities. The static composition method presented in this chapter is both formal and complete. Formal composition constructs and their semantics are defined. It is complete in the sense that the composition is defined on all parts of *ConfiguredService*, not just on service functionality. The primary advantage of formalism and completeness is that complex expressions of composed *ConfiguredServices* can be constructed and subjected to formal analysis of quality properties. Formal analysis is necessary because service expressions are often complex, involving many composition operators, and hard to do by manual inspection at execution time. At the same time not all properties should require formal verification. Moreover, the service-oriented architecture should support formal verification by delegating it to an architectural unit. In Chapter 7, we discuss these issues in some depth, and provide a formal verification approach based on UPPAAL model checker.

5.1 Composition Constructs

The term *compositionality* refers to the ability to compose system specifications, and the term *composition* refers to a specific method for composing sub-systems. The formal manner in which we have defined a *ConfiguredService* makes our service-oriented system composable. Because, we attempt composition at the *ConfiguredService* level. Since *ConfiguredService* formalism is model-based and *compositionality* of model-based specifications have been studied we are justified to claim that *ConfiguredService* specifications are composable. With this basis we discuss composition methods in this section. Inspired by the work [WMA09], we define the composition constructs and informally motivate their meanings by stating their *intended* execution behavior. Services require *resources* at execution time. Lack of adequate resources will have an adverse impact on the quality of service. So, in giving semantics for composite services we assume that resources are available at execution time for executing all services in the composition. Only under such ideal situation a fair semantics can be given.

We also define a graphical notation for each composition construct. They are intended to be used by non-experts. The graphical notations can be transformed into formal service expressions.

A service provider creates a *service expression* involving the names of *ConfiguredServices* and composition constructs. All composition constructs in a service expression have the same precedence, and hence a service expression is evaluated from left to right. To enforce a particular order of evaluation, parenthesis may be used. The result of evaluating a service expression is a *ConfiguredService* and the service provider publishes this result. So, a published service is either *atomic* in the sense that it is not a composed service or it is composite in the sense that it is a composition of at least two atomic services. Note that the atomic *ConfiguredServices* used in producing a composite *ConfiguredService* might not be published by a service provider.

5.1.1 Sequential Composition Construct $\gg$

Given two *ConfiguredServices* A and B, the service expression $A \gg B$ defines a *ConfiguredService* C which is the sequential composition of A and B. The intended execution behavior of the *ConfiguredService* C is the execution behavior of B immediately after the execution of A. That is, service A is to be executed first and its output is to be used in the execution of *ConfiguredService* B, in addition to any input that B may require, in order to fully realize the behavior of C. The service provider who composes $A \gg B$ will publish the service parts of A and B in the service part of C, explicitly making clear the output of A that will be used as the input for B. The contract part of C will meet his business objectives. If we subject the informal and formal descriptions of C to the analyses discussed in Section 4.3 we should guarantee this behavior. In general, the expression $A_1 \gg A_2 \ldots \gg A_k$ denotes the execution of *ConfiguredService* A_{i+1} with the result of execution of A_i as its input, for $i = 1, \ldots, k-1$, in addition to other input that A_{i+1} might need. Figure 7 illustrates the graphical notation for representing sequential composition.

Example 1 *Let A be the service 'airline booking', B be the service 'hotel booking' and C be the service resulting from the sequential composition of A and B. The business motivation for this composition might be that the hotel is a trading partner with the airline, offering special discounts to a specific set of airline customers. This necessitates airline booking confirmation before feeding its output in attempting hotel booking. The service provider, by hosting the composite service, might benefit economically. The consumer will see the service parts for A and B in the published* ConfiguredService *C.*

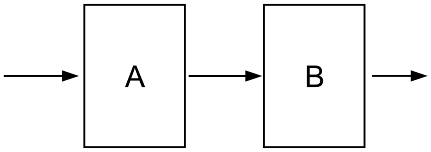


Figure 7: Sequential Composition

5.1.2 Parallel Composition Construct $||$

Given two *ConfiguredServices* A and B, the service expression $A||B$ defines the parallel composition of A and B. The parallel composition $A||B$ service models the simultaneous executions of *ConfiguredServices* A and B. Therefore the resulting behavior of this composite service should be the merging of their individual behaviors in time order. That is, an execution of the composite service should trigger both services A and B to begin at the same instant, and terminate only when both services have finished their executions. The publication of the composite service $A||B$ will include the service parts of A and B. Figure 8 illustrates the graphical notation for representing parallel composition. In general, the evaluation of the expression $A_1 \parallel A_2 \parallel \ldots \parallel A_k$ will create k service execution threads, one for each *ConfiguredService*.

Example 2 *Let A be the service ’airline booking’, B be the service ‘hotel booking’ and C be the parallel composition of A and B. The business motivation for this composition might be that the service provider is a travel agent who can only provide a package of air tickets and hotel reservations. Although the two services are independent from each other they both should be confirmed before the service provider executes C. The consumer should see the service parts of both* ConfiguredServices *A and B in the published* ConfiguredService *C.*

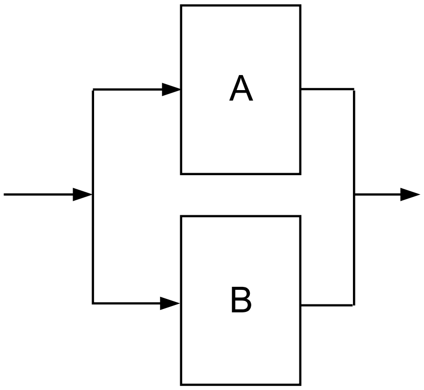


Figure 8: Parallel Composition

5.1.3 Priority Composition Construct $\prec$

Priority construct is very effective in stating the choice of service from a sequence of services such that the chosen service is the earliest service in the sequence that either does not fail or satisfies all user constraints. Given two *ConfiguredServices* A and B, the expression $C = A \prec B$ states (1) $\langle A, B \rangle$ is a sequence of services (ordered), and (2) the service execution of A should be attempted first, and if it succeeds, the service B is to be discarded, and (3) otherwise, the execution of service B should be attempted. So, the behavior of this expression is the behavior of the earliest service in the list of services that can successfully execute. Hence, the publication of C should include the publications of A and B. The meaning of the expression $A_1 \prec \ldots \prec A_k$ is either none of the listed service execute successfully or

- services $A_1, \ldots, A_{j-1}$, for some j, $1 \leq j < k$ fails to execute, and
- A_j can be executed successfully.

In the former case the service expression is null, meaning no service is executed. In the later case the behavior of $A_1 \prec \ldots \prec A_k$ is the behavior of A_j. Figure 9 illustrates our graphical notation for representing priority composition.

Example 3 *A service provider who priority composes $C = A \prec B$, where the* Configured-Service *A is 'booking in airline A', and the* ConfiguredService *B is 'booking in airline B', will publish both A and B and indicate in the service part of C the priority sequence $\langle A, B \rangle$. The business motivation for this composition might be that the service provider gets a higher commission from the first service.*

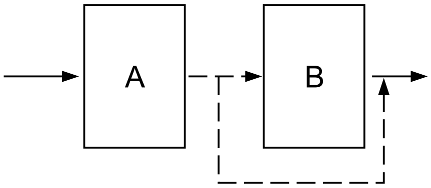


Figure 9: Priority Composition

5.1.4 Composition with No Order Construct $\diamond$

Given two *ConfiguredServices* A and B, the executed behavior of the expression $C = A \diamond B$ is either the behavior of $A \gg B$ or the behavior of $B \gg A$, except that the result output from executing the first service is *not* passed as input to the execution of the second service. That is, their inputs are independent. This does not necessarily mean that the services can be executed in parallel, although the executions may be started simultaneously this is *not imposed*. Therefore, the result of the composition is the set of results produced by the executions of (1) B followed by A, (2) A followed by B, and (3) A and B in parallel. Figure 10 illustrates our graphical notation for representing no order composition. In general, the expression $A_1 \diamond A_2 \diamond \ldots \diamond A_k$ defines the composition of services A_i, $i = 1, k$ when all of them may be executed in no specific order. This composition type is useful when a large random sampling of service orderings is required.

Example 4 *A service provider publishes the composite service obtained by the no order composition of 'A: airline booking service' and 'B: car renting service'. The business motivation for this composition might be that the service provider is a travel agent and he* *can only provide a package of air tickets and car renting. The two services are independent from each other and either service can be executed before the other. The consumer might see the listing of* ConfiguredServices A *and* B *as part of the published service.*

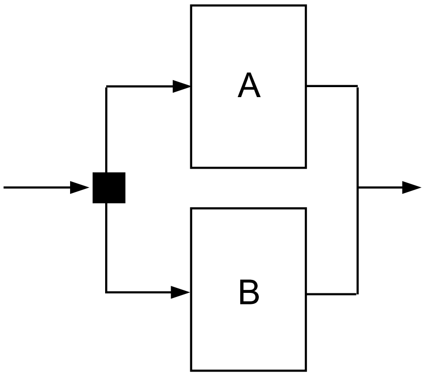


Figure 10: No Order Composition

5.1.5 Nondeterministic Choice Composition Construct $\wr$

Given two *ConfiguredServices* A and B, the service expression $A \wr B$ defines the execution behavior of one of the services to be executed nondeterministically. This meaning is extended for the general case $C = A_1 \wr \ldots \wr A_k$ with k operands. So, the service provider should include the *ConfiguredServices* $A_1, \ldots, A_k$ in the publication of C. Figure 11 illustrates our graphical notation for representing nondeterministic choice composition.

Example 5 *A service provider publishes the composite service obtained by the nondeterministic choice composition of 'airline booking service A' and 'airline booking service B'. The business motivation for this composition might be that the service provider gets the same commission from both services. Hence, both services have the same priority.*

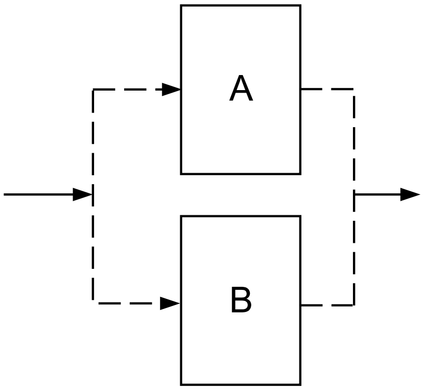


Figure 11: Nondeterministic Composition

5.1.6 Conditional Choice Composition Construct (if-else) $\triangleright$

Given two *ConfiguredServices* A and B, the behavior of the service expression $C = A \triangleright_c B$ is the behavior of A if $\triangleright_c$ evaluates to true, otherwise the behavior of C is the behavior of B. Figure 12 illustrates our graphical notation for representing conditional choice composition. The publication of service C will include the publications of A and B and the condition for selecting one of them.

Example 6 *A service provider publishes the composite service obtained by the conditional choice composition $C = A \triangleright_c B$, where A denotes the 'booking service in airline A', B denotes the service 'booking service in airline B', and $\triangleright_c$ is the condition $DEPARTURE == Montreal$. The business motivation for this composition might be that the service provider gets a higher commission from the first service if the departure is from Montreal and a higher commission from the second service if the departure is from any other city.*

5.1.7 Iteration Composition Construct (while) $\circ$

The behavior of the composition $A_{\circ_c}$ is the iterative accumulative behavior of executing service A as long as the condition $\circ_c$ remains true. Figure 13 illustrates the graphical

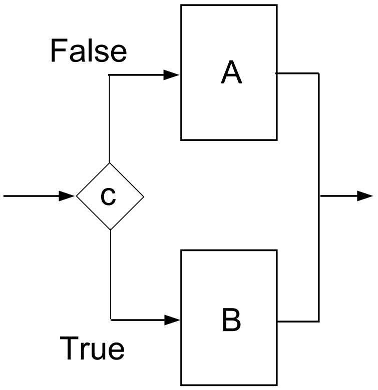


Figure 12: Conditional Composition

notation for representing iteration composition.

Example 7 *A service provider publishes the composite service obtained by the iterative composition of 'hotel booking service' as long as rooms are available. As part of this publication the full* ConfiguredService *description for hotel booking service, and the condition for booking will be included. The contractual details will be comprehensive for available rooms. The business motivation for this composition might be that the service provider gets a higher discount from this hotel than any other hotel. So he prefers to reserve all rooms in this hotel.*

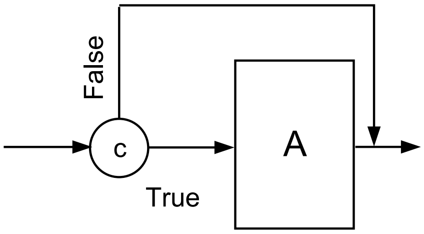


Figure 13: Iteration Composition

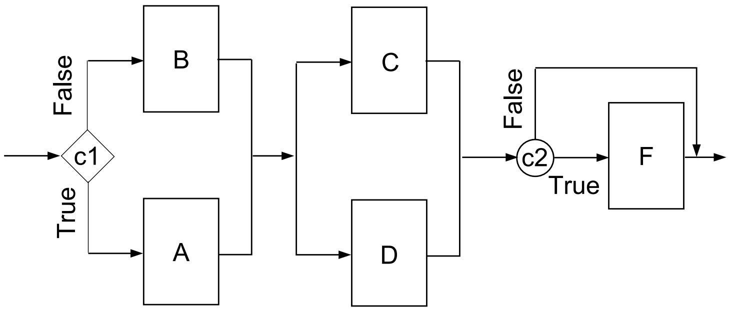


Figure 14: Execution logic of $(A \triangleright_{c1} B) \gg (C||D) \gg F_{\circ_{c2}}$

Example 8 *The execution logic of the composite service* $(A \triangleright_{c1} B) \gg (C||D) \gg F_{\circ_{c2}}$*, shown in Figure 14, is obtained by putting together the execution logics defined above.*

5.2 Semantics of *ConfiguredService* Compositions

Every service provider has a business model. Motivated by the business rules and logic in the model, a service provider will determine the nature of composition for services. We want to emphasize that the *meaning* of a composition primarily rests on the chosen business goals and rules. Consequently, service compositions are very much *unlike* action compositions based purely on preconditions and postconditions. As an example, a service provider may form $A \gg B$ because it is either technically necessary or advantageous in business terms to provide service B following the completion of service A. That is, service B cannot be realized without first executing service A. This is analogous to 'bootstrapping' before invoking any other system function in the domain of computing services. This implies that the precondition for invoking a system function includes the precondition for invoking 'bootstrapping', however it might require more conditions to be met. Moreover, the postcondition of 'bootstrapping' and the postcondition of the system function invoked after that are both observed. In some domains, it might happen that the precondition for invoking service B is exactly the same as the postcondition of the first service A, and is not observable. Only the postcondition of B, after B is completed, may be observable. Given such subtle scenarios, it is hard to give one 'fixed' semantics for service compositions. The semantics given below formalizes the informal explanations given in Section 5.1. By providing a formal semantics for composition constructs we are motivating a theory of composition in which complex service expressions can be meaningfully expressed and interpreted. From the formal representation of a composite *ConfiguredService*, it is possible to generate more than one consistent informal *ConfiguredService* representation. The service provider stands to benefit by this flexibility.

Below we let $A = \langle \sigma_A, \mu_A \rangle$, and $B = \langle \sigma_B, \mu_B \rangle$ denote two *ConfiguredServices*, where $\sigma_A = \langle f_A, \kappa_A, \alpha_A \rangle$, $\sigma_B = \langle f_B, \kappa_B, \alpha_B \rangle$, $\mu_A = \langle \delta_A, l_A, \beta_A \rangle$, $\mu_B = \langle \delta_B, l_B, \beta_B \rangle$, $f_A = \langle g_A, i_A, pr_A, po_A \rangle$, $f_B = \langle g_B, i_B, pr_B, po_B \rangle$, $g_A = \langle n_A, d_A, u_A \rangle$, $g_B = \langle n_B, d_B, u_B \rangle$, $i_A = \langle m_A, q_A \rangle$, $i_B = \langle m_B, q_B \rangle$, $\kappa_A = \langle p_A \rangle$, $\kappa_B = \langle p_B \rangle$, $\delta_A = \langle tr_{cs_A}, tr_{p_A} \rangle$, $\delta_B = \langle tr_{cs_B}, tr_{p_B} \rangle$, $tr_{cs_A} = \langle \rho_A, \epsilon_A, \psi_A, \eta_A \rangle$, $tr_{cs_B} = \langle \rho_B, \epsilon_B, \psi_B, \eta_B \rangle$, $tr_{p_A} = \langle ce_A, pg_A, re_A \rangle$, $tr_{p_B} = \langle ce_B, pg_B, re_B \rangle$, $\beta_A = \langle r_A, c_A \rangle$, and $\beta_B = \langle r_B, c_B \rangle$. For the sake of simplicity we assume that the currency type $cType$ and the unit type $uType$ are the same for all services. The result of a composition is a *ConfiguredService*.

5.2.1 Semantics of Sequential Composition

The sequential composition $A \gg B$ of *ConfiguredServices* A and B is a *ConfiguredService*, expressed as the tuple $\langle \sigma_{A \gg B}, \mu_{A \gg B} \rangle$ whose components are defined below.

- **Service**: $\sigma_{A \gg B} = \langle f_{A \gg B}, \kappa_{A \gg B}, \alpha_{A \gg B} \rangle$

  1. **Function**: $f_{A \gg B} = \langle g_{A \gg B}, i_{A \gg B}, pr_{A \gg B}, po_{A \gg B} \rangle$, $g_{A \gg B} = \langle n_{A \gg B}, d_{A \gg B}, u_{A \gg B} \rangle$, $i_{A \gg B} = \langle m_{A \gg B}, q_{A \gg B} \rangle$, where

$$\begin{array}{lll}
g_{A \gg B}: & & \\
n_{A \gg B} & = & n_A \frown n_B & \text{naming convention} \\
d_{A \gg B} & = & d_A \cup d_B & \text{combine input data parameters} \\
u_{A \gg B} & = & \{u_A, u_B\} & \text{both function addresses are necessary}
\end{array}$$

$$\begin{array}{llll}
i_{A \gg B}: & & & \\
m_{A \gg B} & = & m_A \frown m_B & \text{naming convention} \\
q_{A \gg B} & = & q_A \cup q_B & \text{combine output parameters}
\end{array}$$

$$\begin{array}{llll}
pr_{A \gg B} & = & pr_A \cup (pr_B \setminus po_A) & \text{if } B \text{ requires more constraints} \\
pr_{A \gg B} & = & pr_A & \text{if } B \text{ does not require more constraints} \\
po_{A \gg B} & = & po_A \cup po_B & \text{if } po_A \text{ is not used as an input of } B \\
po_{A \gg B} & = & po_B & \text{if } po_A \text{ is absorbed as an input for } B
\end{array}$$

2. **Nonfunctional Properties**: $\kappa_{A \gg B} = \langle p_{A \gg B} \rangle$ where, $p_{A \gg B} = \langle a_{A \gg B}, cu_{A \gg B}, un_{A \gg B} \rangle$ where $cu_{A \gg B}$ = cu_A = cu_B, $un_{A \gg B}$ = un_A = un_B, and

$$a_{A \gg B} = \begin{cases} a_A + a_B & \text{normal pricing} \\ max\{a_A, a_B\} & \text{promotional} \\ min\{a_A, a_B\} & \text{special sale} \end{cases}$$

3. **Attributes**: $\alpha_{A \gg B} = \alpha_A \cup \alpha_B$

- **Contract**: $\mu_{A \gg B} = \langle \delta_{A \gg B}, l_{A \gg B}, \beta_{A \gg B} \rangle$, where

  1. **Trustworthiness**: $\delta_{A \gg B} = \langle tr_{cs_{A \gg B}}, tr_{p_{A \gg B}} \rangle$, where $tr_{cs_{A \gg B}} = \langle \rho_{A \gg B}, \epsilon_{A \gg B}, \psi_{A \gg B}, \eta_{A \gg B} \rangle$, $tr_{p_{A \gg B}} = \langle ce_{A \gg B}, pg_{A \gg B}, re_{A \gg B} \rangle$ and

     - Safety (timeliness): $\rho_{A \gg B} = \rho_A + \rho_B$.
     - Safety (data): $\rho_{A \gg B} = \rho_A \wedge \rho_B$.
     - Security: $\epsilon_{A \gg B} = \epsilon_A \cup \epsilon_B$.
     - Availability: $\eta_{A \gg B} = \eta_A + \eta_B$.

- Reliability: $\psi_{A \gg B} = \psi_A + \psi_B$.
- ProviderTrust: Given a set s_t of trust values, it should be possible to define $avg(s_t)$, $choose(s_t)$, $glb(s_t)$, and $lub(s_t)$ which respectively computes the average, selects randomly one value, and computes the least and greatest values from the set s_t. Any one of these functions may be used by the service provider in providing ce and re. Each choice has some significance. Choosing avg reflects 'unbiased views of customers', choosing $choose$ reflects a randomly selected customer opinion, choosing glb reflects a conservative estimate, and choosing lub reflects the optimistic opinion of customers. For illustration, we use the function glb. We compute the trust sets as:

$$\begin{aligned}
ce_{A\backslash B} &= \{(a,b) \mid (a,b) \in ce_A, (a,b) \notin ce_B\} \\
&\quad \text{Recommendation given for } A \text{ only} \\
ce_{B\backslash A} &= \{(a,b) \mid (a,b) \notin ce_A, (a,b) \in ce_B\} \\
&\quad \text{Recommendation given for } B \text{ only} \\
ce_{A\cap B} &= \{(a,b) \mid (a,b_1) \in ce_A, (a,b_2) \in ce_B, b = glb(b_1,b_2)\} \\
&\quad \text{Recommendation given for } A \text{ and } B
\end{aligned}$$

Similar sets for re are defined. The trust for the composition $A \gg B$ can be defined for different semantics.

  ∗ *Business Logic: Service A is required for service B.* In this situation the expectation is that those who bought service B should have obtained service A, and hence they bought the service $A \gg B$. That is, the recommendation for B dominates. With this semantics we define

$$\begin{aligned}
ce_{A\gg B} &= ce_{A\cap B} \cup ce_{B\backslash A} \\
re_{A\gg B} &= re_{A\cap B} \cup re_{B\backslash A}
\end{aligned}$$

  ∗ *Business Logic: Those who bought service A are most likely to buy service B.* In this situation buying A is a certainty. Not everyone who bought A may buy B. That is, service recommendation for A dominates. With this semantics we define

$$ce_{A\gg B} = ce_{A\cap B} \cup ce_{A\backslash B}$$

$$re_{A\gg B} = re_{A\cap B} \cup re_{A\backslash B}$$

$*$ *Business Logic: Both services are packaged together*: With this semantics the service provider has to collect the sets ce and re from clients and organizations for the new service.

In all above situations

$$pg_{A\gg B} = pg_A \wedge pg_B$$

2. **Legal Issues**: $l_{A\gg B} = l_A \cup l_B$, defined as the union of the issues of A and B.

3. **Context**: We use the semantics of context union ($\sqcup$) and sub-context ($\sqsubset$), as defined by Wan [Wan06]. These are defined essentially using relational semantics. For *ConfiguredServices* A the context is $\beta_A = \langle r_A, c_A \rangle$. This means that r_A is true in context c_A in order that A may be provided. Once the service A has been provided, the context and rules that are true in that context should be computed. Letting these rules r'_A and the context c'_A, we need to merge them with r_B and c_B, $\beta_B = \langle r_B, c_B \rangle$ to arrive at $\beta_{A\gg B}$. With this rationale, we define $\beta_{A\gg B} = \langle r_{A\gg B}, c_{A\gg B} \rangle$, $r_{A\gg B} = r'_A \cup r_B$, and $c_{A\gg B} = c'_A \sqcup c_B$, the smallest closure of contexts c'_A and c_B. It is expected that $c'_A \sqsubset c_B$ holds for most of the applications, because anything outside of c_B can be ignored.

5.2.2 Parallel Composition Semantics

The parallel composition $A||B$ of the *ConfiguredServices* A and B is a *ConfiguredService*, expressed as the tuple $\langle \sigma_{A||B}, \mu_{A||B} \rangle$ whose components are defined below.

- **Service**: $\sigma_{A||B} = \langle f_{A||B}, \kappa_{A||B}, \alpha_{A||B} \rangle$

1. **Function**: $f_{A||B} = \langle g_{A||B}, i_{A||B}, pr_{A||B}, po_{A||B} \rangle$, $g_{A||B} = \langle n_{A||B}, d_{A||B}, u_{A||B} \rangle$, $i_{A||B} = \langle m_{A||B}, q_{A||B} \rangle$, where

$g_{A||B}$:

$n_{A||B} \;=\; n_A \frown n_B$ naming convention

$d_{A||B} \;=\; d_A \cup d_B$ both input data parameter sets are required

$u_{A||B} \;=\; \{u_A, u_B\}$ both function addresses are required

$i_{A||B}$:

$m_{A||B} \;=\; m_A n \frown m_B$ naming convention

$q_{A||B} \;=\; q_A \cup q_B$ both output data parameter sets will be available

$pr_{A||B} \;=\; pr_A \cup pr_B$ preconditions are mutually disjoint

$po_{A||B} \;=\; po_A \cup po_B$ both postcondition sets are available

2. **Nonfunctional Properties**: $\kappa_{A||B} = \langle p_{A||B} \rangle$, where $p_{A||B} = \langle a_{A||B}, cu_{A||B}, un_{A||B} \rangle$ and:

   $a_{A||B} \;=\; a_A + a_B$

   $cu_{A||B} \;=\; cu_A = cu_B$

   $un_{A||B} \;=\; un_A = un_B$

3. **Attributes**: $\alpha_{A||B} = \alpha_A \cup \alpha_B$

- **Contract**: $\mu_{A||B} = \langle \delta_{A||B}, l_{A||B}, \beta_{A||B} \rangle$, where

  1. **Trustworthiness**: $\delta_{A||B} = \langle tr_{cs_{A||B}}, tr_{p_{A||B}} \rangle$, where $tr_{cs_{A||B}} = \langle \rho_{A||B}, \epsilon_{A||B}, \psi_{A||B}, \eta_{A||B} \rangle$, $tr_{p_{A||B}} = \langle ce_{A||B}, pg_{A||B}, re_{A||B} \rangle$ and

     - Safety (timeliness): $\rho_{A||B} = Max(\rho_A, \rho_B)$.
     - Safety (data): $\rho_{A||B} = \rho_A \wedge \rho_B$.
     - Security: $\epsilon_{A||B} = \epsilon_A \cup \epsilon_B$.
     - Availability: $\eta_{A||B} = Max(\eta_A, \eta_B)$.
     - Reliability: $\psi_{A||B} = Max(\psi_A, \psi_B)$.
     - ProviderTrust: As in the case of $A \gg B$, we calculate the three sets for ce and three trust sets for re. Using them we define $tr_{A||B} = \langle ce_{A||B}, pg_{A||B}, re_{A||B} \rangle$ where,

$$\begin{aligned} ce_{A||B} &= ce_{A\backslash B} \cup ce_{B\backslash A} \cup ce_{A\cap C} \\ pg_{A||B} &= pg_A \wedge pg_B \\ re_{A||B} &= re_{A\backslash B} \cup re_{B\backslash A} \cup re_{A\cap C} \end{aligned}$$

2. **Legal Issues**: $l_{A||B} = l_A \cup l_B$, defined as the union of the issues of A and B.

3. **Context**: $\beta_{A||B} = \langle r_{A||B}, c_{A||B} \rangle$, where $r_{A||B} = r_A \cup r_B$, and $c_{A||B} = c_A \sqcup c_B$.

5.2.3 Priority Composition Semantics

For the priority composition $A \prec B$ of the *ConfiguredServices* A and B the execution of the *ConfiguredService* A first. If the execution is not successful, an execution to B will be attempted. Example semantics is the one based on the satisfaction of context rules at stated contexts. The composition semantics is

- if the service delivery context satisfies the context condition stated in A then the execution of A is attempted and B is ignored;

- if the service delivery context satisfies the context condition stated in B then the execution of B is attempted, and A is ignored;

- otherwise no service execution is attempted

Consequently, the resulting *ConfiguredService* is at most one of $\{A, B\}$.

5.2.4 Composition with No Order Semantics

The composition with no order $A \diamond B$ of *ConfiguredServices* A and B is *ConfiguredService*, expressed as the tuple $\langle \sigma_{A\diamond B}, \mu_{A\diamond B} \rangle$ whose components are defined below.

- **Service**: $\sigma_{A\diamond B} = \langle f_{A\diamond B}, \kappa_{A\diamond B}, \alpha_{A\diamond B} \rangle$

  1. **Function**: $f_{A\diamond B} = \langle g_{A\diamond B}, i_{A\diamond B}, pr_{A\diamond B}, po_{A\diamond B} \rangle$, $g_{A\diamond B} = \langle n_{A\diamond B}, d_{A\diamond B}, u_{A\diamond B} \rangle$, $i_{A\diamond B} = \langle m_{A\diamond B}, q_{A\diamond B} \rangle$, where

$g_{A\diamond B}$:

$$\begin{aligned} n_{A\diamond B} &= n_A \frown n_B && \text{naming convention} \\ d_{A\diamond B} &= d_A \cup d_B && \text{both input sets are required} \\ u_{A\diamond B} &= \{u_A, u_B\} && \text{both addresses are required} \end{aligned}$$

$i_{A\diamond B}$:

$$\begin{aligned} m_{A\diamond B} &= m_A \frown m_B && \text{naming convention} \\ q_{A\diamond B} &= q_A \cup q_B && \text{both output sets are generated} \end{aligned}$$

$$\begin{aligned} pr_{A\diamond B} &= pr_A \cup pr_B \\ po_{A\diamond B} &= po_A \cup po_B \end{aligned}$$

2. **Nonfunctional Properties**: $\kappa_{A\diamond B} = \langle p_{A\diamond B}\rangle$, where $p_{A\diamond B} = \langle a_{A\diamond B}, cu_{A\diamond B}, un_{A\diamond B}\rangle$ and:

$$\begin{aligned} a_{A\diamond B} &= a_A + a_B \\ cu_{A\diamond B} &= cu_A = cu_B \\ un_{A\diamond B} &= un_A = nu_B \end{aligned}$$

3. **Attributes**: $\alpha_{A\diamond B} = \alpha_A \cup \alpha_B$

- **Contract**: $\mu_{A\diamond B} = \langle \delta_{A\diamond B}, l_{A\diamond B}, \beta_{A\diamond B}\rangle$, where

  1. **Trustworthiness**: $\delta_{A\diamond B} = \langle tr_{cs_{A\diamond B}}, tr_{p_{A\diamond B}}\rangle$, where $tr_{cs_{A\diamond B}} = \langle \rho_{A\diamond B}, \epsilon_{A\diamond B}, \psi_{A\diamond B}, \eta_{A\diamond B}\rangle$, $tr_{p_{A\diamond B}} = \langle ce_{A\diamond B}, pg_{A\diamond B}, re_{A\diamond B}\rangle$ and

     – Safety (timeliness): $\rho_{A\diamond B} = t$, where $Max\{\rho_A, \rho_B\} \leq t \leq \rho_A + \rho_B$.

     – Safety (data): $\rho_{A\diamond B} = \rho_A \wedge \rho_B$.

     – Security: $\epsilon_{A\diamond B} = \epsilon_A \cup \epsilon_B$.

     – Availability: $\eta_{A\diamond B} = t$, where $Max\{\eta_A, \eta_B\} \leq t \leq \eta_A + \eta_B$.

     – Reliability: $\psi_{A\diamond B} = t$, where $Max\{\psi_A, \psi_B\} \leq t \leq \psi_A + \psi_B$.

     – ProviderTrust: As in the case of $A \gg B$, we calculate the three sets for ce and three trust sets for re. Using them we define $tr_{A\diamond B} = \langle ce_{A\diamond B}, pg_{A\diamond B}, re_{A\diamond B}\rangle$.

$$
\begin{aligned}
ce_{A\diamond B} &= ce_{A\backslash B} \cup ce_{B\backslash A} \cup ce_{A\cap C} \\
pg_{A\diamond B} &= pg_A \wedge pg_B \\
re_{A\diamond B} &= re_{A\backslash B} \cup re_{B\backslash A} \cup re_{A\cap C}
\end{aligned}
$$

The trust values in sets $ce_{A\cap B}$ and $re_{A\cap B}$ can be calculated as illustrated for the composition $A \gg B$.

2. **Legal Issues**: $l_{A\diamond B} = l_A \cup l_B$, defined as the union of the issues of A and B.
3. **Context**: $\beta_{A\diamond B} = \langle r_{A\diamond B}, c_{A\diamond B}\rangle$, where $r_{A\diamond B} = r_A \cup r_B$, and $c_{A\diamond B} = c_A \sqcup c_B$.

5.2.5 Nondeterministic Choice Composition Semantics

The nondeterministic choice construct will result in choosing one service for execution. Therefore, the semantics of the composition is the semantics of the service selected. For example, $\beta_{A\wr B} = \beta_A$ if A is chosen, otherwise $\beta_{A\wr B} = \beta_B$. In a similar manner we define the other components of $A \wr B$.

5.2.6 Conditional Choice Composition Semantics (if-else)

The conditional composition $A \triangleright_c B$ will result in defining two *ConfiguredServices*. The first *ConfiguredService* is equal to A and an attempt for its execution occurs if the condition c is true. The second *ConfiguredService* is equal B and an attempt for its execution occurs if c is false. The published *ConfiguredService* $A \triangleright_c B$ will include the publications of A and B and the condition c for selecting one of them.

5.2.7 Iteration Composition Semantics (while)

The iterative composition $A\circ_c$ can be defined using a finite number k of sequential composition, as $A \gg A \gg \ldots \gg A$, when the condition $\circ_c$ explicitly contains the iteration constant k (as in for loops). In this case the sequential composition semantics defined above can be used in defining the composition of the *ConfiguredServices*. However when $\circ_c$ is like a 'while loop specifying an invariant' then we need the fixed point semantics $A\circ_c = A \gg A\circ_c$. In general, the service expression A in $A_{\circ_c}$ can itself involve an iteration.

The semantics of evaluating such expressions is derived in a top down manner. The service part of the published *ConfiguredService* contains the functionality of A and the condition c. It defines the nonfunctional properties and trustworthiness guarantees as functions with respect to the number of iterations. For example, if the cost of A is equal to 50$, the published *ConfiguredService* will state that the price is equal to $(50 * n)$$, where is the number of iterations.

5.3 Case Study - Auto Roadside Emergency Service

The Auto Road Emergency Example introduced in Section 4.5 requires a sequential composition of the three *ConfiguredSerivces* RepairShop rs, TowTruck tt, and CarRental cr. In this section, we introduce the composition result using the formal definition presented in Section 4.5 and the composition rules presented in this chapter. The composition is divided into two steps. In the first step, the composition $rs \gg tt$ is calculated. In the second step, the composition $rs \gg tt \gg cr$ is calculated.

5.3.1 Composing $rs \gg tt$

The composite *ConfiguredService* is the tuple $s_{rs \gg tt} = \langle \mu_{rs \gg tt}, \sigma_{rs \gg tt} \rangle$, where the tuple components are explained below.

Service: $\sigma_{rs \gg tt} = \langle f_{rs \gg tt}, \kappa_{rs \gg tt}, \alpha_{rs \gg tt} \rangle$ where,

1. **Function:** $f_{rs \gg tt} = \langle g_{rs \gg tt}, i_{rs \gg tt}, pr_{rs \gg tt}, po_{rs \gg tt} \rangle$ where,

   - **Signature:** $g_{rs \gg tt} = \langle n_{rs \gg tt}, d_{rs \gg tt}, u_{rs \gg tt} \rangle$, where $n_{rs \gg tt} = (ReserveRS\&TT)$ is the name, $d_{rs \gg tt} = \{(CarBroken, bool), (Deposit, double), (CarType, string), (FailureType, string), (RequestTruck, bool), (ShopLoc, location), (CarLoc, location)\}$ are input data parameters, and $u_{rs \gg tt} = (XXXYYY)$ is the address.

   - **Result:** $i_{rs \gg tt} = \langle m_{rs \gg tt}, q_{rs \gg tt} \rangle$, where $m_{rs \gg tt} = (ResultRS\&TT)$ is the name and the set of output data parameters is $q_{rs \gg tt} = \{(HasAppointment, bool),$

$(numberOfHours, int), (ShopLoc, location), (RequestConfi, bool)\}$.

- **Precondition:** $pr_{rs \gg tt} = ((CarBroken == true) \wedge (RequestTruck == true))$.
- **Postcondition:** $po_{rs \gg tt} = ((HasAppointment == true) \wedge (RequestConfi == true))$.

2. **Nonfunctional:** $\kappa_{rs \gg tt} = \langle p_{rs \gg tt} \rangle, p_{rs \gg tt} = \langle a_{rs \gg tt}, cu_{rs \gg tt}, un_{rs \gg tt} \rangle$, where $a_{rs \gg tt} = ((60 * numOfHours) + ((Distance/80) * 100))$ is the cost, $cu_{rs \gg tt} = (dollar)$ is the currency, and $un_{rs \gg tt} = (oneTime)$ is the pricing unit. The distance between the repair shop and the broken car $Distance$ is calculated using the following equation:

$$Distance = \sqrt[3]{(x_{ShopLoc} - x_{CarLoc})^2 + (y_{ShopLoc} - y_{CarLoc})^2 + (z_{ShopLoc} - z_{CarLoc})^2}$$

3. **Attributes:** $\alpha_{rs \gg tt} = \{(name = Garage1), (name = Truck1)\}$.

Contract: $\mu_{rs \gg tt} = \langle \delta_{rs \gg tt}, l_{rs \gg tt}, \beta_{rs \gg tt} \rangle$ where,

1. **Trustworthiness:** $\delta_{rs \gg tt} = \langle tr_{cs}, tr_p \rangle$ where,
   - ServiceTrust: $tr_{cs} = \langle \rho_t \rangle$ where $\rho_t = 3days + 45minutes = 4365minutes.$
   - ProviderTrust: $tr_p = \langle re \rangle$ where $re = \{(CAA, Excellent)\}$.
2. **Legal:** $l_{rs \gg tt} = \{(deposit = 300), (CarType == toyota), (method ==''\ Cash'')\}$.
3. **Context:** $\beta_{rs \gg tt} = \langle r_{rs \gg tt}, c_{rs \gg tt} \rangle$, where $r_{rs \gg tt} = \{(membership == caa)\}$ is the context rule and $c_{rs \gg tt} = \{(Location, (Montreal, Canada))\}$ is the contextual information of the repair shop service provider.

5.3.2 Composing $rs \gg tt \gg cr$

Using the composition result of $rs \gg tt$ and using the sequential composition rules, we can compose $rs \gg tt \gg cr$. The resulting *ConfiguredService* is the tuple $s_{rs \gg tt \gg cr} = \langle \mu_{rs \gg tt \gg cr}, \sigma_{rs \gg tt \gg cr} \rangle$, where the tuple components are explained below.

Service: $\sigma_{rs \gg tt \gg cr} = \langle f_{rs \gg tt \gg cr}, \kappa_{rs \gg tt \gg cr}, \alpha_{rs \gg tt \gg cr} \rangle$ where,

1. **Function:** $f_{rs \gg tt \gg cr} = \langle g_{rs \gg tt \gg cr}, i_{rs \gg tt \gg cr}, pr_{rs \gg tt \gg cr}, po_{rs \gg tt \gg cr} \rangle$ where,

    - **Signature:** $g_{rs \gg tt \gg cr} = \langle n_{rs \gg tt \gg cr}, d_{rs \gg tt \gg cr}, u_{rs \gg tt \gg cr} \rangle$, where $n_{rs \gg tt \gg cr} = (ReserveRS\&TT\&CR)$ is the name, $d_{rs \gg tt \gg cr} = \{(CarBroken, bool), (Deposit, double), (CarType, string), (FailureType, string), (RequestTruck, bool), (CarLoc, location), (ShopLoc, location), (NeedCar, bool), (CarSize, string), (StartDate, date), (EndDate, date)\}$ are input data parameters, and $u_{rs \gg tt \gg cr} = (XXXYYYZZZ)$ is the address.

    - **Result:** $i_{rs \gg tt \gg cr} = \langle m_{rs \gg tt \gg cr}, q_{rs \gg tt \gg cr} \rangle$, where $m_{rs \gg tt \gg cr} = (ResultRS \&TT\&CR)$ is the name and the set of output data parameters is $q_{rs \gg tt \gg cr} = \{(HasAppointment, bool), (numberOfHours, int), (ShopLoc, location), (Re questConfi, bool), (HasCar, bool), (ConfNum, string)\}$.

    - **Precondition:** $pr_{rs \gg tt \gg cr} = ((CarBroken == true) \wedge (RequestTruck == true) \wedge (NeedCar == true))$.

    - **Postcondition:** $po_{rs \gg tt \gg cr} = ((HasAppointment == true) \wedge (RequestConfi == true) \wedge (HasCar == true))$.

2. **Nonfunctional:** $\kappa_{rs \gg tt \gg cr} = \langle p_{rs \gg tt \gg cr} \rangle, p_{rs \gg tt \gg cr} = \langle a_{rs \gg tt \gg cr}, cu_{rs \gg tt \gg cr}, un_{rs \gg tt \gg cr} \rangle$, where $a_{rs \gg tt \gg cr} = ((60 * numOfHours) + ((Distance/80) * 100) + ((EndDate - StarDate) * 30))$ is the cost, $cu_{rs \gg tt \gg cr} = (dollar)$ is the currency, and $un_{rs \gg tt \gg cr} = (oneTime)$ is the pricing unit.

3. **Attributes:** $\alpha_{rs \gg tt \gg cr} = \{(name = Garage1), (name = Truck1), (name == Renatal)\}$.

Contract: $\mu_{rs \gg tt \gg cr} = \langle \delta_{rs \gg tt \gg cr}, l_{rs \gg tt \gg cr}, \beta_{rs \gg tt \gg cr} \rangle$ where,

1. **Trustworthiness:** $\delta_{rs \gg tt \gg cr} = \langle tr_{cs}, tr_p \rangle$ where,

    - ServiceTrust: $tr_{cs} = \langle \rho_t, \epsilon_{cr} \rangle$ where $\rho_t = 3days + 45minutes = 4365minutes$ and $\epsilon_{cr} = \{(encryption = 128)\}$.

- ProviderTrust: $tr_p = \langle re \rangle$ where $re = \{(CAA, Excellent)\}$.

2. **Legal:** $l_{rs \gg tt \gg cr} = \{(deposit == (300 + 200)), (CarType == toyota)$, $(TowPay\ Method ==''\ Cash'')$, $(RentalPayMethod ==\ ''CreditCard'')\}$.

3. **Context:** $\beta_{rs \gg tt \gg cr}$ = $\langle r_{rs \gg tt \gg cr}, c_{rs \gg tt \gg cr} \rangle$, where $r_{rs \gg tt} = \{(membership == caa)\}$ is the context rule and $c_{rs \gg tt} = \{(Location, (Montreal, Canada))\}$ is the contextual information of the repair shop service provider.

5.4 Summary

In this Chapter, we discussed composability and static compositions for *ConfiguredServices*. We defined several composition constructs, explained informally the meaning of service expressions, and gave a formal semantics for compositions. We illustrated these concepts for composing the services in the auto roadside emergency service.

Chapter 6

Composition Verification

A service composition consists of multiple interacting *ConfiguredServices* that provide a functionality to meet a specific set of requirements. It is essential to verify that the functional behavior of the service composition meets the published functionality of the service composition while taking into consideration the nonfunctional, legal and contextual conditions. The verification is necessary regardless of the composition method. In other words, the composition resulting from static service composition or dynamic service composition should be verified. Since both compositions have the same set of operators the verification approach presented in this section is valid for both static and dynamic compositions.

Instead of defining a new verification tool to verify the service composition we follow a transformation approach. In this approach, a formally defined service composition can be automatically transformed into a model understood by an available verification tool that can then be used to perform the formal verification. The goal in our research is to use different verification tools in order to verify a wide range of properties and target different kinds of systems. This is because different verification tools differ in their requirements and abilities. In this section, we define the transformation rules to generate a model that can be verified using UPPAAL [BDL04a] model checking tool. The rest of this section is structured as follows. First, we present a brief account of the model checking tool UPPAAL. Second, the rules to transform a service composition into a UPPAAL model are presented. Third, the verification process is discussed. Finally, an example is presented to illustrate the verification process.

6.1 A Brief Review of UPPAAL

UPPAAL [BDL04b] is a mature tool for the modeling, simulation and verification of real-time systems. It is designed to verify systems which can be modeled as networks of *timed automata (TA)* extended with integer variables, structured data types, and channel synchronization. A TA is a finite-state machine extended with clock variables. It can be formally defined as a tuple $\langle L, L_0, K, A, E, I \rangle$, where L is a set of *locations* denoting the states, L_0 is the *initial* state, K is a set of *clocks*, A is a set of *actions* that cause transitions between locations, E is a set of *edges*, $E \subseteq L \times A \times B(K) \times 2^k \times L$, where $B(K)$ is the set of data and time constraints that restrict the transitions and 2^k is the set of clock initializations to set clocks whenever required, I is a set of *invariants*, where $I : L \rightarrow B(K)$ is a function that assigns time constraints to clocks. UPPAAL extends the definition of TA with additional features. Below are some of these features that are relevant to our goal.

- **Templates**: TAs are defined as templates with optional parameters. Parameters are local variables that are initialized during template instantiation in system declaration.

- **Global variables**: Global variables and user defined functions can be introduced in a global declaration section. Those variables and functions are shared and can be accessed by all templates.

- **Binary synchronization**: Two TA can have a synchronized transition, caused by an event, when both move to new state at the same time when the event occurs. An event that causes synchronous transition is defined as a channel, a UPPAAL data type. A channel can have two directions: input (labeled with ?) and output (labeled with!).

- **Committed Location**: Time is not allowed to pass when the system is in a committed location. If the system state includes a committed location, the next transition must involve an outgoing edge from the committed location.

- **Expressions**: There are three main types of expressions (1) *Guard expressions*, which are evaluated to Boolean and used to restrict transitions, they may include clocks and state variables, (2) *Assignment expressions*, which are used to set values of clocks and variables, and (3) *Invariant expressions*, which are defined for locations and used to specify conditions that should be always true in a location.

- **Edges**: Edges denote transitions between locations. An edge specification consists of four expressions (1) *Select*, which assigns a value from a given range to a defined variable, (2) *Guard*, an edge is enabled for a location if and only if the guard is evaluated to true, (3) *Synchronization*, which specifies the synchronization channel and its direction for an edge, and (4) *Update*, an assignment statements that reset variables and clocks to required values.

UPPAAL can be used by users to specify a checking formula that contains a set of properties. The checking formula can be a combination of the following [BY04]: (1) `A[]`φ, which means φ *will invariantly happen*, (2) `E<>`φ, which means φ *will possibly happen*, (3) `A<>`φ, which means φ *will always happen eventually*, (4) `E[]`φ, which means φ *will potentially always happen*, and (5) φ`-->` ψ, which means φ *will always lead to* ψ. Where φ and ψ are Boolean expressions defined on locations, integer variables, and clocks constraints. The properties that can be checked using UPPAAL are [BDL04b]:

- *Reachability:* UPPAAL can check whether or not it is possible to reach a certain location. It also checks whether or not there is a deadlock in the system.

- *Safety:* UPPAAL can check whether or not anything bad will ever happen, declared in UPPAAL as something good is always true.

- *Liveness:* UPPAAL can check whether or not something will happen eventually.

6.2 Transforming the Service Composition into UPPAAL TA

This section presents the rules for transforming a service composition into a UPPAAL TA. Let $S = \{s_1, ..., s_n\}$ be the set of *ConfiguredServices* to be composed. Let Υ be the composition expression defining the composition, and $SC = \langle S, \Upsilon, \mu, \sigma \rangle$ be the resulting composition. Let $TA = \langle L, L_0, K, A, E, I \rangle$ be the definition of a UPPAAL TA, where L is a set of *locations* denoting the states, L_0 is the *initial* state, K is a set of *clocks*, A is a set of *actions* that cause transitions between locations, E is a set of *edges*, and I is a set of *invariants*. The transformation rules will construct $T = \{ta_1, ..., ta_n\}$, a set of UPPAAL templates. The first step is to define the following in the global declaration section in UPPAAL.

1. Two channel variables are defined for each s_i. The first represents the request and the second represents the response.

2. A Boolean variable is defined for every precondition and input parameter in SC and assigned to *true*. These variables are used to verify if preconditions and input parameters exist before execution.

3. A Boolean variable is defined for every postcondition and output parameter in SC and assigned to *false*. These variables are used to verify if postconditions and output parameters exist after execution.

4. A typed variable is defined for every parameter in SC. The type can be any simple type, such as `int`, or a structured data type.

5. The following variables of type `double` are defined and assigned to 0 for each composition flow:

   - *PathPrice*, which represents the total price of the composition flow.
   - *PathAvailability*, which represents the availability of the composition flow.

- *PathReliability*, which represents the reliability of the composition flow.
- *PathTime*, which represents the safety time guarantee of the composition flow.

6. Boolean variables representing the elements of the legal issues are defined. These variables are used in defining the Legal issues as Boolean statements.

7. A UPPAAL structure that represents the contextual information of the service requester is defined. The structure contains dimensions and associated tag values.

6.2.1 Transformation Rules

The transformation rules are divided into two sets. The first set defines the rules to transform an individual *ConfiguredService* into a TA. The second set defines the rules to transform the composition flow into a TA.

Rules to Transform a *ConfiguredService* Each *ConfiguredService* can be mapped to a UPPAAL template in a one to one manner. A *ConfiguredService* $s_i = \langle \mu_i, \sigma_i \rangle$ is mapped to a template $ta_i = \langle L_i, L_{0i}, K_i, A_i, E_i, I_i \rangle$. Following are the transformation rules to generate ta_i for each s_i.

1. For each ta_i create two locations $L_i = \{l_1, l_2\}$, and set the first location as the initial state $L_{0i} = \{l_1\}$.

2. Create two edges $E_i = \{e_1, e_2\}$ in ta_i, with edge e_1 directed from l_1 to l_2 and edge e_2 directed from l_2 to l_1.

3. Define an action for each s_i and add it to A_i.

4. Add to edge e_1 the following expressions:

   (a) Add to guard the condition that all s_i preconditions are equal to true.

   (b) Add to guard the condition that all s_i input parameters are available.

   (c) Add to guard the condition that the s_i contextual rules are satisfied.

(d) Add to guard the condition that the s_i legal rules are satisfied.

(e) Add to Sync the channel variable corresponding to s_i request and follow it with ?.

5. Add to edge e_2 the following expressions:

   (a) Add to update the statement that assign all s_i postconditions variables to true.

   (b) Add to update the statement that assign all s_i output parameters variable to true.

   (c) Add to Sync the channel variable corresponding to s_i responses and follow it with !.

Rules to Transform a Composite Service The next step is to generate the main TA that maps to the composition execution flow. Before generating this TA, the composition flow should be flattened to contain only sequential composition constructs $\gg$. In essence, every composition flow can be flattened into multiple composition flows of *ConfiguredServices* that are executed sequentially. This makes sense, as services are executed by a centralized execution engine and this engine can only execute *ConfiguredServices* sequentially. For some services the execution engine might *wait* for a response before executing another service, for others it might send multiple requests without waiting for the response. Flattening a parallel composition construct $A||B$ will result in two composition flows. The first contains $A\dot{\gg}B$ and the second contains $B\dot{\gg}A$, where $\dot{\gg}$ indicates that the sequential construct resulted from flattening a parallel construct. This indication will be essential when transforming to TA as it indicates no *wait*. Flattening each conditional choice construct $A \rhd_c B$, priority construct $A \prec B$, or nondeterministic choice construct $A \wr B$ will result into two composition flows, where the first contains A and the second contains B. Flattening each no order composition construct $A\Diamond B$ will result in two composition flows. The first contains $A \gg B$ and the second contains $B \gg A$. Flattening the iteration construct $A\circ_c$ will also result into two composition flow, where the first does not contain A and the second contains one or many A.

Example 9 *The composition $(A \triangleright_{c1} B) \gg (C||D) \gg F_{\circ_{c2}}$ can be flattened into 8 composition flows, where X_c indicates that X is associated with condition c. These are: (1) $A_{c1} \gg C \grave{\gg} D$, (2) $A_{c1} \gg C \grave{\gg} D \gg F_{c2} ... \gg F_{c2}$, (3) $A_{c1} \gg D \grave{\gg} C$, (4) $A_{c1} \gg D \grave{\gg} C \gg F_{c2} ... \gg F_{c2}$, (5) $B_{\neg c1} \gg C \grave{\gg} D$, (6) $B_{\neg c1} \gg C \grave{\gg} D \gg F_{c2} ... \gg F_{c2}$, (7) $B_{\neg c1} \gg D \grave{\gg} C$, and (8) $B_{\neg c1} \gg D \grave{\gg} C \gg F_{c2} ... \gg F_{c2}$.*

The main TA will contain an idle state. For each flattened composition flow, a path of states is created in the main TA starting from this idle state according to the following rules.

1. For each *ConfiguredService* create two states. The first represents the request for the *ConfiguredService* and the second represents the completion of the execution.

2. For each *ConfiguredService*, if it contains a safety time constraint, create a new clock and add the timing constraint as an invariant on the location. Exception: if the sequential construct resulted from parallel flattening $X \grave{\gg} Y$, only add the invariant to the state with the highest time constraint of X and Y, and make the other state a committed state.

3. For each *ConfiguredService* create two edges. The first connects the state representing the previous *ConfiguredService* in the flow, except for the first *ConfiguredService* where it connect idle state, to the first state defined in rule 1. The second connects the first state to the second state of rule 1.

4. If the *ConfiguredService* is associated with a condition (conditional choice or iteration condition), add this condition as a guard statement on the first edge of rule 3.

5. If the *ConfiguredService* has a safety data conditions, add this condition as a guard statement on the first edge of rule 3.

6. If the *ConfiguredService* has a price, add to the second edge of rule 3 an update statement that adds the price to the path price variable.

7. If the *ConfiguredService* has an availability property, add to the second edge of rule 3 an update statement that adds the availability to the path availability variable.

8. If the *ConfiguredService* has a reliability property, add to the second edge of rule 3 an update statement that adds the reliability to the path reliability variable. Exception: if the sequential construct resulted from parallel flattening, the update statement is only added to the edge with the highest reliability value.

6.3 Verification Steps

Using UPPAAL editor, the *ConfiguredServices* and their composition are specified as UPPAAL templates following the automatic transformation rules defined in Section 6.2. With UPPAAL verifier we can verify the following properties on the templates.

- **Context**: The context rules are not contradictory, and are met for each *ConfiguredService*.

- **Functionality**: The behavior of the composition is correct with respect to functionality, which includes verifying.

  - The preconditions of each participating *ConfiguredService* are met before invocation.

  - The input parameters of each participating *ConfiguredService* are available before invocation.

  - The composition generates the required postconditions and output parameters.

- **Nonfunctional properties**: The behavior of the composition is correct with respect to nonfunctional properties, which includes verifying.

  - The composition price is greater than or equal the price of any possible execution flow.

- **Trustworthiness properties:** The behavior of the composition is correct with respect to the trustworthiness claims, which includes verifying.
  - The composition safety time constraint is greater than or equal the time constraint of any possible execution flow.
  - The composition availability time is greater than or equal to the availability time of any possible execution flow.
  - The composition reliability value is greater than or equal to the reliability value of any possible execution flow.
- **Legal issues**: The legal rules are not contradictory, and are met for each *ConfiguredService*.

6.4 Case Study - Auto Roadside Emergency Services

Applying the transformation rules defined above to the service composition $RepairShop \gg TowTruck \gg CarRental$ introduced in Section 5.3, the composition is transformed into 4 TA's mapped to 4 UPPAAL templates, a template for each *ConfiguredService* and a template for the composition flow. The TA mapped to the *ConfiguredService* RepairShop is $ta_{rs} = \langle L_{rs}, L_{0rs}, K_{rs}, A_{rs}, E_{rs}, I_{rs} \rangle$, as seen in Figure 6.15(a), where the tuple components are explained below

- The set of locations is $L_{rs} = \{idle, RepairShopProcessing\}$ and the initial location is $L_{0rs} = idle$.
- The set of clocks is $k_{rs} = \{k_1\}$ and the set of invariants is $I_{rs} = \{(k_1 \leq 3)\}$.
- The set of actions is $A_{rs} = \{ScheduleApt, AptConfirmed\}$.
- The set of edges is $E_{rs} = \{(idle{-}RepairShopProcessing), (RepairShopProcessing{-}idle)\}$.

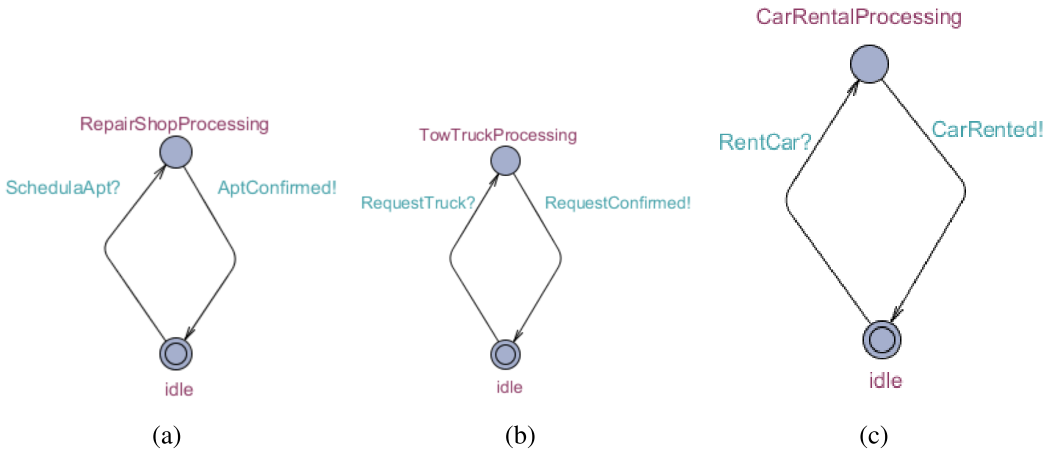


Figure 15: a) RepairShop TA, b) TowTruck TA, and c) CarRental TA

- The edge connecting ’idle’ to ’RepairShopProcessing’ has the following statements, where ’parameterB’ refers to the variable indicating the availability of the parameter ’parameter’:

  - *Guard*: `(RequesterContext.membership==1)&&(CarBroken==true) &&(carType==toyota)&&carTypeB&&failureTypeB&&DepositB`.

  - *Synchronous*: `ScheduleApt?`.

  The edge connecting `RepairShopProcessing` to `idle` has the following statements:

  - *Update*: `HasAppointment =true,NumOfHoursB=true,Deposit=Deposit +300,ShopLocB=true`.

  - *Synchronous*: `AptConfirmed!`.

The TAs mapped to the *ConfiguredServices* TowTruck and CarRental are created in the same manner. Figure 16 shows the generated main TA. In this example, we assume that the pricing unit is uniform and that the price for the *ConfiguredServices* Repair Shop, Tow Truck and Car Rental, are 300$, 100$ and 180$ respectively. UPPAAL is used to verify

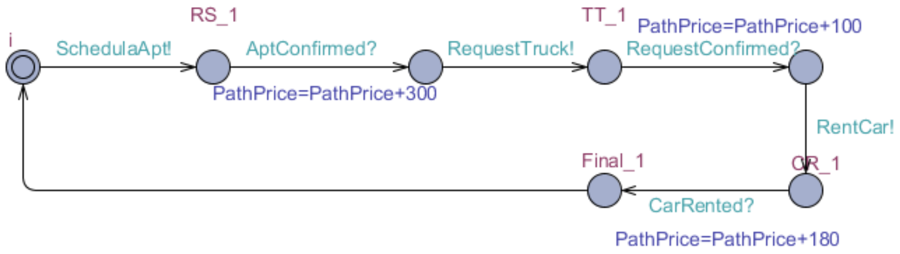


Figure 16: Resulted Main TA

several properties listed below. The notations `M.i` and `M.Final_1` are used to denote the initial and final states of the TA `M`.

- The composition does not contain any contradiction and can be executed. If the UPPAAL statement `E<> M.Final_1` is verified it implies that it is possible to reach the final state of the composition flow. Reaching the final state indicates that all conditions are met and no contradictions exist.

- The context rules are met. For each context rule an UPPAAL verification condition is generated and verified. For example, `A[] M.i imply RequesterContext.age>=21` is the condition to be verified to assert that the requester is older than 21. Here, `RequesterContext` is the UPPAAL structure holding the contextual information of the service requester.

- The composition input parameters are defined before executing the composition flow. For example, `A[] M.i imply failureTypeB` is the condition to be verified in order to assert that the car `failureType` parameter is available before execution. Here, `failureTypeB` is a Boolean variable representing the availability of the parameter `failureType`.

- The composition output parameters are defined after executing the composition flow. For example, `A[] M.i imply !NumOfDaysB` is the condition to be verified in order to assert that the number of days needed to fix the car are not known before executing the composition. The statement `A[] M.i imply !NumOfDaysB`, if verified, asserts that the number of days is known after executing the composition.

The parameter NumOfDaysB is a Boolean variable representing the availability of the parameter NumOfDays.

- The preconditions are met before executing the composition and the postconditions are met after. For example, A[] M.i imply NeedCar==true will have to be verified to assert that the precondition “NeedCar” is true at the initial state.
- The composition of nonfunctional properties are correct. For example, A[] M.Final_1 imply firstPathPrice <= 600 will have to be verified to assert that the price of the composite service is less than 600, where 600 is specified as the price of the service composition.
- The composition result of the legal rules are correct. For example, A[] M.Final_1 imply 400>=Deposit will have to be verified to assert that the deposit is less than 400, if the legal rule states that “The service requester should deposit 400 before requesting the service composition”.

6.5 Summary

The significant contribution of this chapter is the set of rules for transforming *ConfiguredServices* and their compositions to UPPAAL templates and using its model checking checker to verify certain properties in the composite services. The entire process can be automated. A tool has been implemented towards this purpose. The transformation process and the tool design has been inspired by the work of Mohammad [Moh09], however there are some significant differences between the two approaches. In [Moh09] safety and security properties are verified, with no regard to context. In our tool, context-dependent verification of safety and security properties are done. In addition, nonfunctional properties, legal rules, availability, and reliability are verified in the specified context. The proof of correctness given in [Moh09] is easily extendable to prove the correctness of transformation discussed in this chapter.

Chapter 7

FrSeC

This chapter first gives a motivation for *the formal framework for providing context-dependent services (FrSeC)* design and its essential features. Second, it introduces the main components of *FrSeC*. Third, a detailed account of each *FrSeC* component is presented. Fourth, the interface of each component is presented. Fifth, a comprehensive service interaction scenario supported by *FrSeC* is presented. Sixth, the adaptability features supported by *FrSeC* are explained. Finally, the *FrSeC* description of the auto roadside emergency service case study is described.

7.1 Motivation and Features

The provision of trustworthy context-dependent services requires a specific set of features, such as support for context information, support for trustworthiness specification, and support for trustworthy interactions. From the extensive literature review presented in Chapter 2, it is clear that currently no framework exists to support all these features. This is what motivated us to introduce the *formal framework for the provision of context-dependent services (FrSeC)* which supports these features. The following set of *FrSeC* features are necessary for processing context-dependent service requests and providing trustworthy services.

1. **Inclusion of contextual information**: The contextual information is essential at four

stages. These are (1) service publication stage, where context information is a necessary ingredient in constraining service contracts, (2) service request stage, where a service request should state precisely the context in which the service is desired to be delivered, (3) selection and planning stage, where contextual information in a service query will be equated against the contexts in published services, and (4) service execution stage, where the context condition will be verified at service execution context.

2. **Trusted transactions**: Service requesters and service providers may remain anonymous during service provision; however it is the responsibility of the provision framework to ensure trustworthy transactions. That is, a service may be provided only by a certified service provider, and a service may be obtained only by an authorized service requester. In addition, transactions should respect contracts and other legal requirements that are imposed on service providers and service requesters.

3. **Inclusion of nonfunctional and trustworthiness properties**: A nonfunctional property governs the quality of service, viewed both as a product and process. At service specification level, service providers specify nonfunctional and trustworthiness properties. In formulating a service query, service requesters specify their required nonfunctional and trustworthiness properties. During selection and composition, the nonfunctional and trustworthiness properties specified by providers and requesters will be equated with each other to find the best match.

4. **Dynamic selection**: The increased number of services available and the frequent dynamic updates to services make it hard for service requesters to select services at design time. The service provision framework should be designed to allow service requesters specify the requirements with the full knowledge that some service bindings may occur only at run time.

5. **Dynamic planning**: Planning at service selection phase is static, while planning at service provision phase is dynamic. During static planning, service selection requires

match-making between services announced by service providers and service requests from service requesters. Service compositions at this phase are static. At service execution time the contextual information and other user-centric requirements might have changed. With the increased number of services and the increased composition complexity, it is difficult to foresee such changes and have all service composition preplanned in a static manner.

6. **Semantic support**: Semantic information is essential at three stages. At service specification stage, services decorated with semantics will enable a best-match service discovery. At service request stage, requesters can formulate their requirements more precisely. At planning stage, both domain knowledge and the semantic information are essential for match making, planning, and composition.

7. **Fault-tolerance support**: If a service fails or becomes unavailable at run time, the service provision framework should recover from this failure by selecting alternative services.

8. **Use of formal methods**: Formalism is necessary to (1) verify the interaction between services by making sure there are no incompatible behaviors between services in a composition, (2) achieve correct automatic composition by verifying that the composition satisfies the requirements of the requester, and (3) check the conformance of requester requirements and the contracts of the services being provided.

9. **Replanning**: At run time, the contextual information of the service consumer and requester might change. The provision framework should support a replanning process to generate a new plan that best satisfies the requirements in the new context.

10. **Negotiation Support**: Service requesters might not find an exact match to their requirements in published services. They might request some modification to existing service contracts. Hence, a negotiation mechanism is essential.

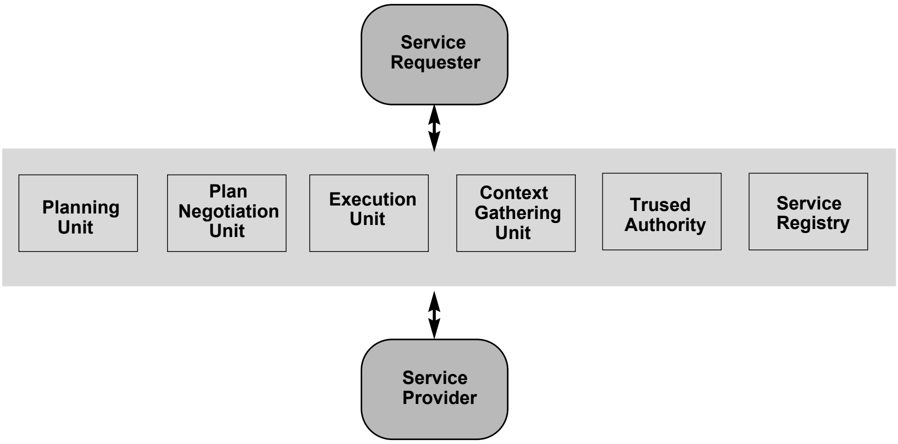


Figure 17: *FrSeC* Components

7.2 FrSeC Components

This section presents the components of *FrSeC*, which collectively have all the features enumerated earlier. Figure 17 shows the components and the features they support.

- **Service Requester (SR)**: It is the entity that is requiring a service. It represents the client side of the interaction. It can be an application or another service. *SR is to fulfill the features 1, 2, 3, 6, 8, 9 and 10.*

- **Service Provider (SP)**: It is the entity that provides an implementation of a service specification. SP publishes service descriptions as *ConfiguredServices* on registries to enable automated discovery and invocation. It is also responsible for defining static service compositions and publishing the resulted composite services. *SP is to fulfill the features 1, 3, 4, 6, 8 and 10.*

- **Context Gathering Unit (CGU)**: Contextual information from related sensors are received and processed at a CGU. *FrSeC* contains at least three such units. One unit gathers the contextual information related to SR and passes it to SR. The second unit gathers contextual information related to SP and passes it to SP. The third unit gathers contextual related to service execution and passes it to EU which might share it with PU. CGU can also detect any change in contextual information. *CGU is to fulfill the* *features 1 and 9.*

- **Planning Unit (PU)**: It is responsible for (1) matching the SP query with available *ConfiguredSerivces*, (2) ranking candidate *ConfiguredServices* according to the SP requirements, and (3) composing services dynamically. *PU is to fulfill the features 1, 2, 3, 4, 5, 6, 7, 8 and 9.*

- **Service Registry (SRe)**: It is responsible for enabling the publication and discovery of services. It also provides semantic definitions for domain specific concepts. *SRe is to fulfill the features 1, 2, 3, 4, 5, 6, 7, 8 and 9.*

- **Plan Negotiation Unit (PNU)**: It acts as a mediator between SR and SP. Its two main functions are (1) finalizing the contracts between SR and SP if no negotiation necessary, and (2) mediating the negotiation between SR and SP, if necessary. After the service contract is agreed upon, it will make sure the SP is still available and is able to provide the required service. *PNU is to fulfill the feature 10.*

- **Execution Unit (EU)**: It is responsible for executing a selected plan. The execution process will include communicating with the SPs involved in the plan by sending service *requests* and obtaining service *responses*. It is also responsible for respecting the privacy rules between the SRs and SPs. *EU is to fulfill the features 2, 4, 5, 7 and 9.*

- **The Trusted Authority (TA)**: The TA is responsible for (1) providing SRs and SPs with certificates (tokens) that allow them to access SRe where the certificate type depends on the legal and contextual information of the SR or SP, (2) enabling the analysis of services before publication and after execution, and (3) the verification of service compositions. *TA is to fulfill the feature 2 and 8.*

7.3 Details of FrSeC Components

Formal notations are necessary for precise communication among software architects and developers. The semantic domain behind the formal notation will help to disambiguate the behavior of architectural elements and help the developer formally assert that certain properties are true in the system under development. The rest of this section presents a detailed formal discussion of the main elements and interactions of *FrSeC* shown in Figure 18.

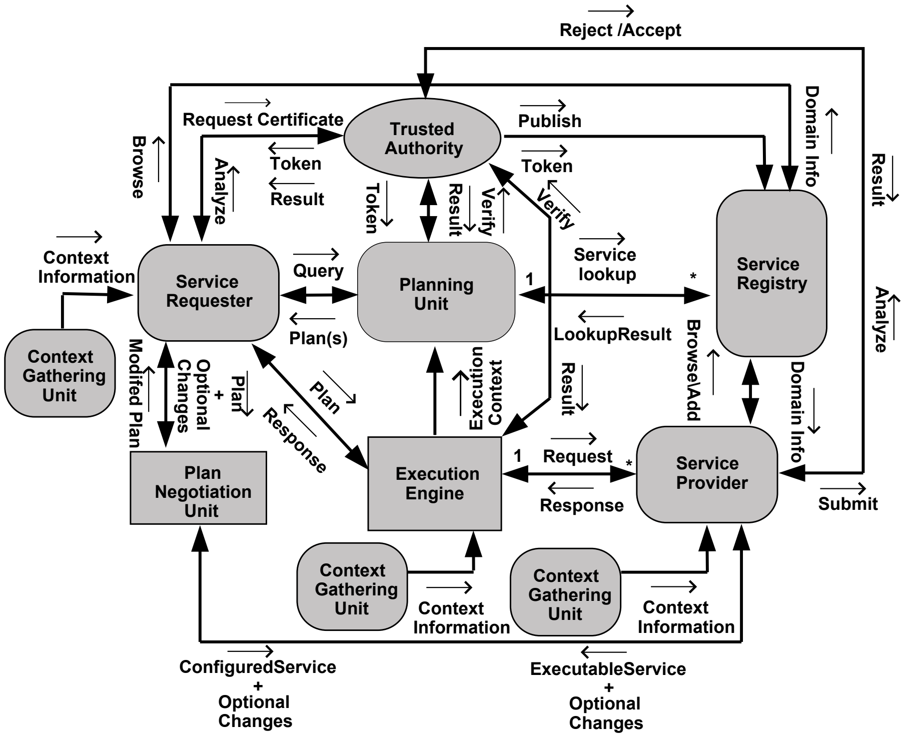


Figure 18: *FrSeC* Architecture

7.3.1 Service Registry (SRe)

The SRe is one of the main elements of the *FrSeC* framework. In this section, we are dealing with the SRe as a black box which provides interfaces to accept services definitions, to request the creation of domains, functions and parameters, and to query about services. The internal process and implementations of the SRe are outside the scope of this thesis.

Before discussing the details of the SRe it is essential to emphasis that the SRe is not intended to host services, they rather contain the service descriptions from SPs who wish to make them public. Hence, the SRe is different from web stores, such the Apple Store. The roles of the SRe are listed below.

- It publishes *ConfiguredServices* defined by SPs.
- It provides a methodology for providing domain knowledge and semantic information that can be used by SPs, SRs and the PU.
- It uses *Role Based Access Control (RBAC)* [FK92] to regulate access to the domain knowledge. Thus, the domain information is protected. Only authorized SRs and SPs can access the parts to which they have received clearance.

The SRe is built and structured in a specific manner to perform its three main roles. It is controlled by a centralized SRe administrator that will ensure the *correctness*, *consistency* and *security* of information in SRe. By correctness we mean that the structure and constraints of the SRe are preserved. For example, a functionality cannot be associated with multiple domains and the administrator should ensure that such thing never happen. Consistency means that the SRe does not contain any contradiction or redundancy. As an example, redundancy arises when the same domain is defined with multiple types. Security means that the SRe elements are accessed only by authorized SRs. The structure of the SRe itself will add semantic meaning to the functionalities provided by the *ConfiguredServices*. For example, the structure defines the relationship between domains and functionalities. This relationship will add semantic meaning to the functionalities by associating it with specific domain.

The SRe is built with the input received from SPs, who can ask the SRe administrator to add domain knowledge to the SRe. This activity includes defining a new domain or a new functionality under an existing domain. The SRe does not aim to define the concepts themselves, but aims to structure the domain knowledge in a useful way for SPs, SRs and the PU. The SRe assumes that the concepts are already well defined by SPs when they perform the analysis of their respective domains. Below is a discussion of the SRe structure.

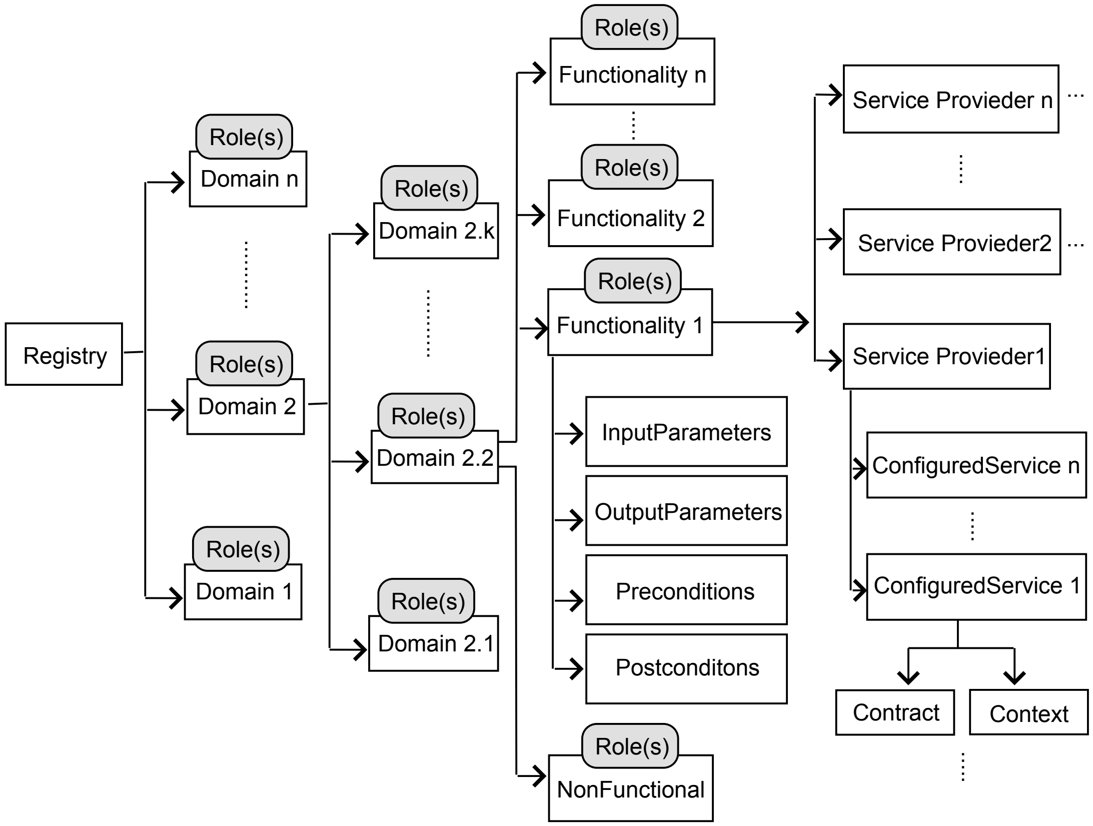


Figure 19: Service Registry

Registry Structure

This section introduces the proposed structure for the SRe. The proposed structure will ensure (1) a simplified browsing by SRs and SPs, (2) a structure that highlights the semantic relation between the different domains and functionalities, and (3) a secure access by SRs and SPs with proper permissions. The SRe is structured as a tree and can be visualized as in Figure 19. The root of this tree is the registry node, which consists of multiple domains defined as its children. That is, the registry node J is a set $\{D_1, D_2, ..., D_n\}$ where D_i represents a domain. A domain node represents a specific domain of knowledge, such as transportation, health care or tourism. A functionality node represents a function that is provided in this domain, such as “reserve a car in a tourism domain”. Domain nodes can have children of type domain or functionality but not both.

A domain node that does not have any child domain node is called a leaf domain node. Only a leaf domain node can have functionality nodes as children, because such a node is the most specific domain for which the functionality is valid. A leaf domain can also have nonfunctional nodes as children. These nonfunctional nodes represent the nonfunctional properties that are associated with this domain. In general, a leaf domain $\hat{D}_i$ may have one or more functionalities and one or more nonfunctional properties. It is represented as the tuple $\langle SF, NF, \Phi \rangle$ where SF is the set of service functionalities defined as part of this domain, NF is the set of nonfunctional properties related to this domain and $\Phi : \{SF_1, SF_2, \ldots, SF_N\} \rightarrow \mathcal{P}\{NF_1, NF_2, ..., NF_n\}$ is a function that associates each functionality with a subset of the functional properties associated with this domain. If $\Phi(SF_i) \neq \emptyset$ then every $NF \in \Phi(SF_i)$ is a nonfunctional property of SF_i.

One of the goals of the SRe is to control the access of the knowledge to the entities that have the required permission. To achieve this goal, we will use *Role based access control (RBAC)* [FK92] mechanism. RBAC is an approach to restrict the access to system parts to a set of specific authorized users. The main concepts in RBAC are *user*, *group*, *role*, and *privilege*. A group defines a set of related users. A user can be individual or belong to one or more groups. A role defines a security responsibility that a user or a group of users can take in the system. A privilege defines a permission to access a domain node or a functionality node. A role comprises many privileges. A privilege can be assigned to many roles. Domain nodes and functionalities nodes of the SRe have restricted access. The SRe will assign roles to its domain and functionalities nodes. Also, the TA provides SRs with certificates indicating their role. If the role in a certificate matches the role associated with a domain node or functionality node then the SR bearing this certificate is allowed to access such node.

Example 10 *Figure 20 shows a simple SRe structure. This SRe consists of three domains Domain1, Domain2 and Domain3. Domain1 does not have any child domain so a functionality can be added to this domain. On the other hand, Domain2 has Domain3 as a child. A* *functionality cannot be added to Domain2 as a child, but a functionality can be added as a child to Domain3.*

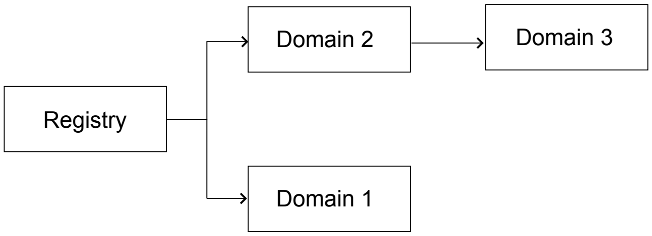


Figure 20: A Service Registry Example

A service functionality can have children nodes representing SPs, parameters, preconditions and postconditions. The children nodes of each SP node are *ConfiguredService* nodes that provide this functionality. A SP can be associated with multiple functionalities. A functionality can be associated with multiple SPs. Each SP node has one or more *ConfiguredService* as its children. The parameters represent the set of data parameters used by the *ConfiguredServices* to provide this functionality. This set is divided into input parameters and output parameters. The precondition represents the minimum set of conditions that should be true before invoking any of the *ConfiguredServices* that provide this functionality. The postcondition represents the maximum set of conditions that are guaranteed to be true after any *ConfiguredService* is invoked.

Formally, a service functionality is defined as the tuple $SF_i = \langle SP, \Lambda, pr_{sf_i}, po_{sf_i} \rangle$, where $SP = \{sp_1, sp_2, ..., sp_n\}$ is the finite set of SPs who provide this service functionality, $\Lambda_{sf} = \{\lambda_i, \lambda_2, ..., \lambda_n\}$ is the finite set of parameters, pr_{sf_i} is the minimum set of preconditions, and po_{sf_i} is the maximum set of postconditions. The set Λ_{sf} represents the union of all the sets of parameters used by the different *ConfiguredServices*, and is the disjoint union of a set of input parameters Λ_{sf_input} and a set of output parameters Λ_{sf_output}, such that $\Lambda_{sf} = \Lambda_{sf\_input} \cup \Lambda_{sf\_output}$. A SP node is defined as the tuple $SP = \langle S, id \rangle$ where $S = \{s_1, s_2, ..., s_n\}$ is the list of *ConfiguredServices* associated with this SP, and id is the SP identification represented as a string.

7.3.2 Service Requester (SR)

The SR is an idealization of a ‘real service requester’ in the system. All real service requesters will behave exactly according to the SR behavior discussed in this section.

To be able to select and invoke a service that meets its requirements, a SR should initiate a discovery process. The discovery process includes *service query, service matching,* and *service ranking*. First, SR defines his requirements in the service query. Second, the query is matched with available *ConfiguredServices* by PU. Third, PU ranks available candidate *ConfiguredServices*. Fourth, SR selects a *ConfiguredService* from the set of ranked *ConfiguredServices*. The novelty of the discovery process supported by *FrSeC* is two-fold. First, the discovery process takes into consideration legal requirements and context conditions together with functional and nonfunctional requirements. Second, depending on the requirements of the SR, *FrSeC* supports the two types of queries *traditional style* and *buffet style*. The rest of this section discusses service query and matching. Service ranking is discussed separately as part of the PU.

Traditional Style Query

In traditional style discovery, the SR has a clear idea about the requirements. But, the semantic information necessary to define the query is missing. Hence, SR accesses SRe to get the domain knowledge which will help in defining the query. The query process can be defined in the following steps:

1. SR sends a request to the TA for a certificate to access SRe.

2. TA provides the certificate depending on the legal and contextual information of SR.

3. SR, with the help of the certificate, browses SRe to gain domain knowledge.

4. SRe provides SR with domain knowledge, such as available domains and their associated functionalities.

5. SR uses this domain knowledge to construct the query and sends it to PU.

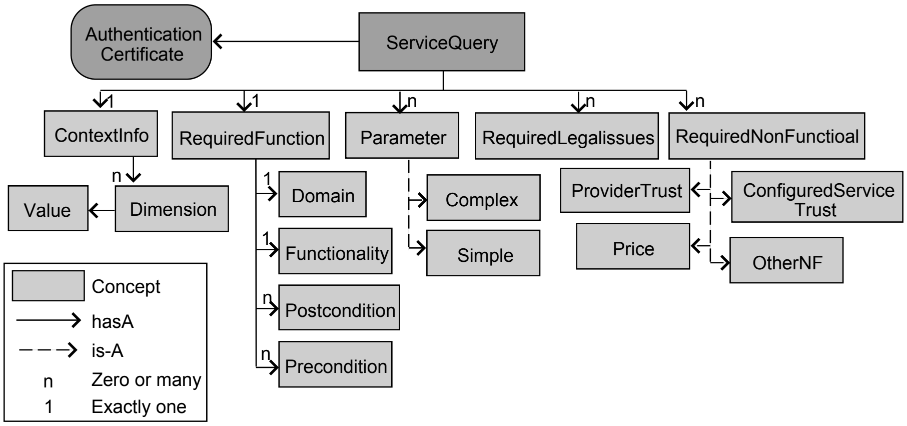


Figure 21: Exact-match Query

6. PU defines and sends service lookups to SRe.

7. The service lookup result is then used by PU to match the query with available services.

8. PU defines the query result (plan) and sends it to SR with its feedback if necessary.

Traditional style discovery can be either *exact-match* discovery or *weighted-match* discovery as discussed below.

Exact-match Discovery: The requester is demanding an exact match to the requirements stated in the query. The candidate *ConfiguredServices* should be able to guarantee all the requirements. The exact-match query, as shown in Figure 21, consists of the five main parts *required function, required nonfunctional properties, required legal issues, consumer contextual information,* and *authentication certificate*. The query also contains the set of parameters that it needs. This set is a subset of the parameters associated with the functionality it chose when accessing SRe. The required nonfunctional properties are a subset of the nonfunctional properties associated with the functionality defined in SRe. The three following definitions formalize an exact-match query.

Definition 12 *An exact-match query q_e is defined as $q_e = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda} \rangle$, where $\hat{f}$ is a query required function, $\hat{\kappa}$ is the nonfunctional requirement, $\hat{l}$ is the legal rules requirements, $\hat{c}$ is the contextual information of the service consumer, E is the authentication* *certificate and $\hat{\Lambda}$ is the set of parameters SR can provide or understand. The formal definitions of context information, legal rules and parameters are identical to the definitions in Section 4.2.*

Definition 13 *The function is defined as $\hat{f} = \langle \hat{pr}, \hat{po}, \hat{D}, \hat{SF} \rangle$, where $\hat{pr}$ is the set of preconditions of the required function, $\hat{po}$ is the set of postconditions of the required function, $\hat{D} : string$ is the associated domain as defined in SRe and $\hat{SF} : string$ is the functionality as defined in SRe. The formal definitions of precondition and postcondition are identical to the one given in Section 4.2.*

Definition 14 *The nonfunctional property is defined as $\hat{\kappa} = \langle \hat{\rho}, \hat{\epsilon}, \hat{\psi}, \hat{\eta}, \hat{p}, \hat{tr} \rangle$, where $\hat{\rho}$ is the required safety guarantee, $\hat{\epsilon}$ is the required security guarantee, $\hat{\psi}$ is the required availability guarantee, $\hat{\eta}$ is required the reliability guarantee, $\hat{p}$ is the maximum price required and $\hat{tr}$ is the required provider trust guarantee. The formal definition of each nonfunctional property is identical to the definition given in Section 4.2.*

Example 11 *If a SR is attempting an exact-match query for the repair shop functionality defined in Chapter 4, the query could be defined as $q_e = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda} \rangle$ where:*

- $\hat{f} = \langle \hat{pr}, \hat{po}, \hat{D}, \hat{SF} \rangle$, *where* $\hat{pr} = (CarBroken == true)$, $\hat{po} = (HasAppointment == true)$, $\hat{D} = (CarDomain)$, *and* $\hat{SF} = (RepairShopFunctionality)$.
- $\hat{\kappa} = \langle \hat{p} \rangle$, *where* $\hat{p} = \langle \hat{a}, \hat{cu}, \hat{un} \rangle$, $\hat{a} = (50)$, $\hat{cu} = (dollar)$ *and* $\hat{un} = (hour)$.
- $\hat{l} = \{(deposit = 500)\}$.
- $\hat{c} = \{(membership == caa)\}$.
- $\hat{\Lambda} = \{(CarBroken, bool), (deposit, double), (CarType, string), (failureType, string)\}$

Weighted-match Discovery: A weighted-match discovery is initiated when the SR states the requirements and gives weights that should be assigned to them. The expectation of SR is that the best matched services, that might not be exact matches, will be given. That is, the *ConfiguredServices* received by the SR do not have to match all the stated requirements. This situation arises because the SR is unsure that all his requirements have equal importance.

When stating the query the requester assigns a weight, representing the priority, with every property requirement. A higher weight indicates a higher priority. SR can also state *exact* property to indicate that an exact match is necessary for this particular property. Stating the weight is valid for the elements of the required function, nonfunctional requirements and the required legal rules. With respect to contextual information, SR can state more than one possible set of contextual information. As an example, the context information for service delivery can be either the service be delivered at home or at office. Each contextual information will be assigned a weight to indicate the preference of the requester. In our further discussion we assume that the assigned weights belong to the set {*Low, BelowAverage, Average, AboveAverage, High, Exact*}, in which the values are listed in strictly increasing order of priority.

Definition 15 *A weighted-match query is defined as* $q_w = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda}, \Xi \rangle$*, where* $\hat{f}$*,* $\hat{\kappa}$*,* $\hat{l}$*,* $\hat{c}$*,* E *and* $\hat{\Lambda}$ *are defined as in the traditional query, and* $\Xi : (x \in \{Low, BelowAverage, Average, AboveAverage, High, Exact\}) \rightarrow (y \in \{\hat{pr}, \hat{po}, \hat{\rho}, \hat{\epsilon}, \hat{\psi}, \hat{\eta}, \hat{p}, \hat{tr}, \hat{l}, \hat{c})\}$ *is a function that assign weights to the elements of the weighted-match query.*

Example 12 *Adding weights to the query defined in Example 11 the weighted-match style query will be defined as* $q_w = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda}, \Xi \rangle$ *where* $\hat{f}$*,* $\hat{\kappa}$*,* $\hat{c}$*,* $\hat{l}$*,* E *and* $\hat{\Lambda}$ *are defined as in Example 11, and* $\Xi = \{((CarBroken == true), \textbf{\textit{Exact}}), ((HasAppointment = = true), \textbf{\textit{Exact}}), (\hat{p}, \textbf{\textit{High}}), ((deposit = 500), \textbf{\textit{Average}})\}$.

The matching process in weighted-match discovery considers all possible *ConfiguredServices* even if some properties are not satisfied. All candidate *ConfiguredServices* will be included in the matching result *ServiceType*, with the exception of the *ConfiguredServices* that do not provide a match for a requirement with $Exact$ weight.

Buffet Style Query

A buffet style query is one which should exactly match a *ConfiguredService*. The SR browses the SRe before formulating such a query. We may assume that the SR feels that a *ConfiguredService* matches his requirements. So, a buffet style query is processed as follows:

1. SR sends a request to the TA for a certificate to access SRe.
2. TA provides the certificate depending on the legal and contextual information of SR.
3. SR, with the help of the certificate, browses SRe for available *ConfiguredServices*.
4. SRe provides SR with high level information about the set of available *ConfiguredServices*.
5. SR defines the query in terms of a specific *ConfiguredService* and sends it to PU.
6. PU will access the SRe to get the complete information about the required *ConfiguredService*.
7. SRe will verify that SR has the required authentication to use the required *ConfiguredService*.
8. PU defines the query result (plan) to include the complete *ConfiguredServices* information and sends it to SR with any feedback if necessary.

No matching process is necessary in buffet style, because the SR is querying only *ConfiguredServices*. As a consequence, the definition of buffet style query, shown in Figure 22, consists of the three main parts *required ConfiguredService, consumer contextual information,* and *authentication certificate*. The following two definitions formalize a buffet style query.

Definition 16 *A buffet style query is defined as* $q_b = \langle \hat{cs}, \hat{c}, E, \hat{\Lambda} \rangle$*, where* $\hat{cs}$*,* $\hat{c}$*,* E*, and* $\hat{\Lambda}$ *are respectively the required* ConfiguredService*, the contextual information, the authentication certificate and the set of parameters SR can provide. These are defined as in Section 4.2.*

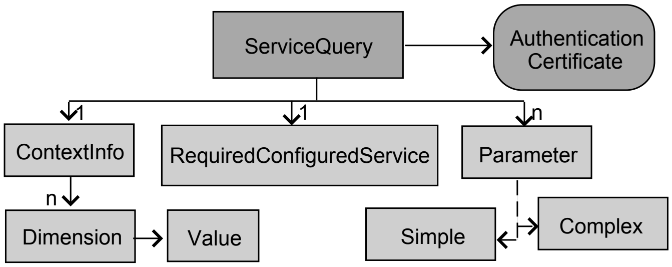


Figure 22: Buffet style Query

Example 13 *If SR is attempting a buffet style query for the* ConfiguredService *Repair-Shop defined in Figure 2, the query will be defined as* $q_b = \langle \hat{cs}, \hat{c}, E, \hat{\Lambda} \rangle$*, where* $\hat{cs} = s_{rs}$*.* $\hat{c} = \{(membership == caa)\}$*, and* $\hat{\Lambda} = \{(CarBroken, bool), (deposit, double), (CarType, string), (failureType, string)\}$

7.3.3 Planning Unit (PU)

The PU defines the service plan that can satisfy a query. A *plan* consists of either a single *ServiceType* or multiple *ServiceTypes*. A *ServiceType* is a set of *ConfiguredServices* that satisfy the set of requirements. A *ServiceType* includes alternative services that can be used in case of a service failure. In addition to fault tolerance, *ServiceTypes* provide services that are useful for adaptation in different contexts.

In order to make a plan the PU should match the requester query with available *ConfiguredSerivces*, ranking candidate *ConfiguredServices* according to the requester requirements, and form service compositions dynamically whenever demanded by the SR. We discuss dynamic compositions separately in Chapter 8. Below we discuss service matching and service ranking.

Service Matching

The PU will receive *ConfiguredServices* from the SRe, as part of the *Service lookup* for the Service Query. For exact-match query, the PU will produce a *ServiceType* in which all *ConfiguredServices* exactly match the requirements defined in the service query. This is achieved through the four matching stages:

1. the functionality in the query must equal the functionality in the *ConfiguredService*,
2. the set of nonfunctional requirements stated in the query must be a subset of the set of nonfunctional properties in the *ConfiguredService*,
3. no legal rule specified in the query must contradict a legal rule specified in the *ConfiguredService*, and
4. the contextual information given in the query must make the *ConfiguredService* context rules true.

For a weighted-match discovery, *ConfiguredServices* that partially match some properties may be considered. Only those *ConfiguredServices* that do not provide a match for a requirement with $Exact$ weight will not be included in the *ServiceType*. A property assigned an $Exact$ weight by a SR is a mandatory requirement for the SR.

For a traditional query, whether exact matching or weighted matching, Algorithm MATCH, presented below, computes the *ServiceType*. It filters out *ConfiguredServices* that do not meet any of the requirements. If *ConfiguredService* is not filtered out then it is a candidate *ConfiguredService*. In this algorithm we use $A \Rightarrow B$ to indicate that property A satisfies property B, or property A implies property B.

Algorithm MATCH

```
INPUT: Traditional Style Query "SQ" and the set of ConfiguredServices "CS".
OUTPUT: A ServiceType "ST".
Create a new empty ServiceType "ST".
for $cs \in CS$ do
  if $cs.precondition.priority == EXACT$ then
    if !($cs.precondition \Rightarrow SQ.precondition$) then
      ignore cs;
    end if
  end if
```

```
**if** $cs.postcondition.priority == EXACT$ **then**
  **if** !($cs.postcondition \Rightarrow SQ.postcondition$) **then**
    ignore cs;
  **end if**
**end if**
**if** $cs.parameters \nsubseteq SQ.parameters$ **then**
  ignore cs;
**end if**
**for** $sq_{nf} \in SQ.nonfunctional$ **do**
  **if** $sq_{nf}.priority == EXACT$ **then**
    **if** !($cs.nonfunctional \Rightarrow sq_{nf}$) **then**
      ignore cs;
    **end if**
  **end if**
**end for**
**for** $sq_{tr} \in SQ.trustworthiness$ **do**
  **if** $sq_{tr}.priority == EXACT$ **then**
    **if** !($cs, trustworthiness \Rightarrow sq_{tr}$) **then**
      ignore cs;
    **end if**
  **end if**
**end for**
**for** $sq_{le} \in SQ.legal$ **do**
  **if** $sq_{le}.priority == EXACT$ **then**
    **if** !($cs.legal \Rightarrow sq_{le}$) **then**
      ignore cs;
    **end if**
  **end if**
**end for**
```

```
        if !($SQ.contextInfo \Rightarrow cs.contextRule$) then
            ignore cs;
        end if
        Add cs to ST
    end for
    return ST;
```

In case of buffet style query, no matching is necessary as the SR has already selected his required *ConfiguredService*.

Service Ranking

For each buffet style query the result of the service lookups is exactly one *ConfiguredService*. Hence, no ranking is necessary. For traditional style query the result of the service lookups and the matching process is multiple candidate *ConfiguredServices*. In exact match queries, *ConfiguredServices* included in the *ServiceType* are in the *order* they were discovered. That is, a *ConfiguredService* A that was discovered earlier than *ConfiguredService* B was discovered will precede it in the *ServiceType* list. For weighted-match discovery, *ConfiguredServices* are included in decreasing order of importance in the *ServiceType*. That is, the service that appears first in the *ServiceType* provides the best match to the query, the service following it provides the next best, and, the last listed service has the least match with the query. Consequently, ranking of services is essential for weighted-match queries. The ranking process can be defined in the following 3 steps.

Step 1: Form Priority Vector The consumer uses the ordered set of priorities $P_c = \{$*Low = 1, BelowAverage = 2, Average = 3, AboveAverage = 4, High = 5, Exact = 6*$\}$, selects a priority to a property that is desired in the *ConfiguredService*, and assigns it to that property. Assume that every *ConfiguredService* has n properties. The PU constructs the vector Q_p, as in Equation 1, where $p_j \in P_c$ is the weight of property j as defined by the consumer. The property j can be a nonfunctional property, a trustworthiness property or a legal rule. If property j is a quantitative property, such as *price* or *shipping time*, the consumer also specifies a value v_j that it considers fit.

$$Q_p = [p_1, p_2, \ldots, p_n] \tag{1}$$

Step 2: Construct Weight Matrix Assume that m *ConfiguredServices* are given to the PU by the consumer. By using the priority vector constructed in Step 1, the values v_j specified by the consumer for quantitative properties, and the service properties included in the i^{th} *ConfiguredService*, the weights w_{ij} are computed, as described in the following two steps. The weight w_{ij} denotes the PU assigned weight for property j in the *ConfiguredService* i.

- *Qualitative properties* A qualitative property j has no 'numerical value' associated with it in the *ConfiguredService*. Trustworthiness properties with no associated numerical values or a legal rule with no numerical values are examples of this kind. For a property j of this type, the consumer cannot specify a numerical value but only state whether or not it is desired. So, the weight w_{ij} is set to 1 if the consumer desires are met in *ConfiguredService* i, and is set to 0 if the property is not met in *ConfiguredService* i.

- *Quantitative properties* Nonfunctional properties such as price, and trustworthiness properties such as legal rules involving numerical values (discounts, penalties), availability or time-safety properties are of this kind. For each quantitative property j of this type, the weight w_{ij} is calculated according to Equation 2, where v_j is the required property value as defined by the consumer and x is the actual property value stated in *ConfiguredService* i.

$$w_{i,j} = \begin{cases} 1 & \text{if } x \leq v_j \\ 1 - (\frac{x - v_j}{2v_j - v_j}) = 2 - \frac{x}{v_j} & \text{if } v_j < x < 2v_j \\ 0 & \text{if } x \geq 2v_j \end{cases} \tag{2}$$

In Equation 2, if the value stated in the *ConfiguredService* i is more than double the required value of that property specified by the consumer, the weight w_{ij} is set to 0. If the value stated in the *ConfiguredService* i is less than the required value of the consumer the weight is set to 1. If the value stated in the *ConfiguredService* i lies between the required value and double the required value of the consumer the weight is chosen proportional to how close the actual value is to the required value.

From the computed weights the weight matrix for the m *ConfiguredServices* is constructed, as shown in Equation 3. In this matrix coloum i represents the weights of the properties in the i^{th} *ConfiguredService*, and row j represents the weights of j^{th} property in the different *ConfiguredServices*.

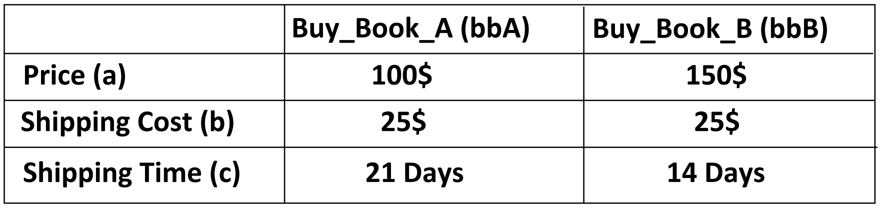
$$CS_w = \begin{bmatrix} w_{11} & w_{21} & .. & w_{m1} \\ w_{12} & w_{22} & .. & w_{m2} \\ .. & .. & .. & .. \\ w_{1n} & w_{2n} & .. & w_{mn} \end{bmatrix} \tag{3}$$

Step 3- Calculate Weights for Ranking Services A single numerical value for each *ConfiguredService* is computed using Equation 4. The vector W contains the weights of the candidate *ConfiguredServices*. These weights are used to rank the *ConfiguredServices* in decreasing order of their weights.

$$W = Q_p \times CS_w \tag{4}$$

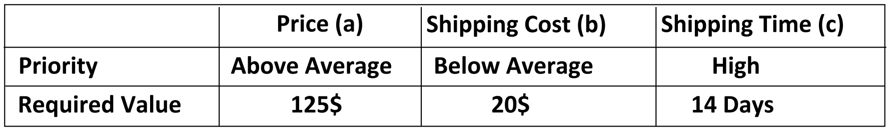
Example 14 *The two* ConfiguredServices Buy_Book_A *and* Buy_Book_B*, shown in Table 9 sell the same book, however they differ in their nonfunctional and trustworthiness properties. Both have the same set of legal rules. The consumer sets a priority and indicates her preferences for the properties through numerical values, as shown in Table 10.*

The priority vector is $Q_p = \begin{bmatrix} AboveAverage, & BelowAverage, & High \end{bmatrix}$*. In numbers,* $Q_p = \begin{bmatrix} 4, & 2, & 5 \end{bmatrix}$*. The weight matrix for the two* ConfiguredServices *is computed*



	Buy_Book_A (bbA)	Buy_Book_B (bbB)
Price (a)	100$	150$
Shipping Cost (b)	25$	25$
Shipping Time (c)	21 Days	14 Days

Table 9: Buy_Book ConfiguredServices



	Price (a)	Shipping Cost (b)	Shipping Time (c)
Priority	Above Average	Below Average	High
Required Value	125$	20$	14 Days

Table 10: Consumer Requirements and Priority

using Equations 3 and 2.

$$CS_w = \begin{bmatrix} w_{bbA,a} & w_{bbB,a} \\ w_{bbA,b} & w_{bbB,b} \\ w_{bbA,c} & w_{bbB,c} \end{bmatrix}$$

where, $w_{bbA,a} = 1$*,* $w_{bbB,b} = 1$*,* $w_{bbB,c} = 1$ *and,*

$$w_{bbB,a} = 2 - \frac{150}{125} = 0.8$$

$$w_{bbA,b} = 2 - \frac{25}{20} = 0.75$$

$$w_{bbA,c} = 2 - \frac{21}{14} = 0.5$$

The ranking vector is computed using Equation 4:

$$W = \begin{bmatrix} 4 & 2 & 5 \end{bmatrix} \begin{bmatrix} 1 & 0.8 \\ 0.75 & 1 \\ 0.5 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 10.3 \end{bmatrix}$$

Hence, the ConfiguredService *Buy_Book_A is ranked second, while the* ConfiguredService *Buy_Book_B is ranked first.*

7.3.4 Plan Negotiation Unit (PNU)

After receiving the plan(s) from the PU, the SR selects the most appropriate plan for him. The selection process takes into consideration the ranking of the plans and other properties that might be of interest for the SR. The selected plan is then passed to PNU which is responsible for two main functions.

1. The SR is satisfied with the selected plan. In this case, the PNU will communicate with all participating SPs in the plan, in order to ensure that participating *ConfiguredServices* are still available.
2. The selected plan, although the best match, is not an exact match to the requirements of the SR. Since the SR might not know the identity of SPs, the PNU acts as a mediator and negotiates with the SP on behalf of the SR.

In case of a need for negotiation, the SR selects the *ConfiguredService* that needs negotiation. The requester also specifies the changes required in the selected *ConfiguredService*. In the negotiation, a change to the *Service* section of the *ConfiguredService* is not allowed. Changes are allowed only in the *Contract* section of the *ConfiguredService*. Hence, the specified changes by the SR can only be in the *Contract*. The contract modification syntax is discussed in Section 4.4. The PNU will communicate with the SP of the *ConfiguredService*. It will pass the required changes. The service provider will decide to accept, reject or modify the changes. This will be send back to the PNU which will pass it to the SR. The SR will review the negotiation result and either accepts or rejects the new modifications. This process continues until an agreement has been reached. And then the SR passes the *ConfiguredService* to the EU for execution.

7.3.5 Service Provider (SP)

A service provider creates *ConfiguredServices*, and have them certified by the TA before they are published in the SRe. SPs usually provide a number of *ConfiguredServices* that have the same functionality, but with differing contracts. The functionality in a *ConfiguredService* may be either *simple* or *composite*. Motivated by business goals, a service provider creates composite services using the methods discussed in Chapter 5. A *ConfiguredService* is *atomic*, in the sense that the service(s) in it cannot be decomposed into simpler services.

Service Publication

A SP will follow the following sequence of steps to publish the *ConfiguredService*. Figure 23 shows this sequence as a flowchart. Before attempting these steps the SP is required to obtain the necessary authentication certificate from the TA.

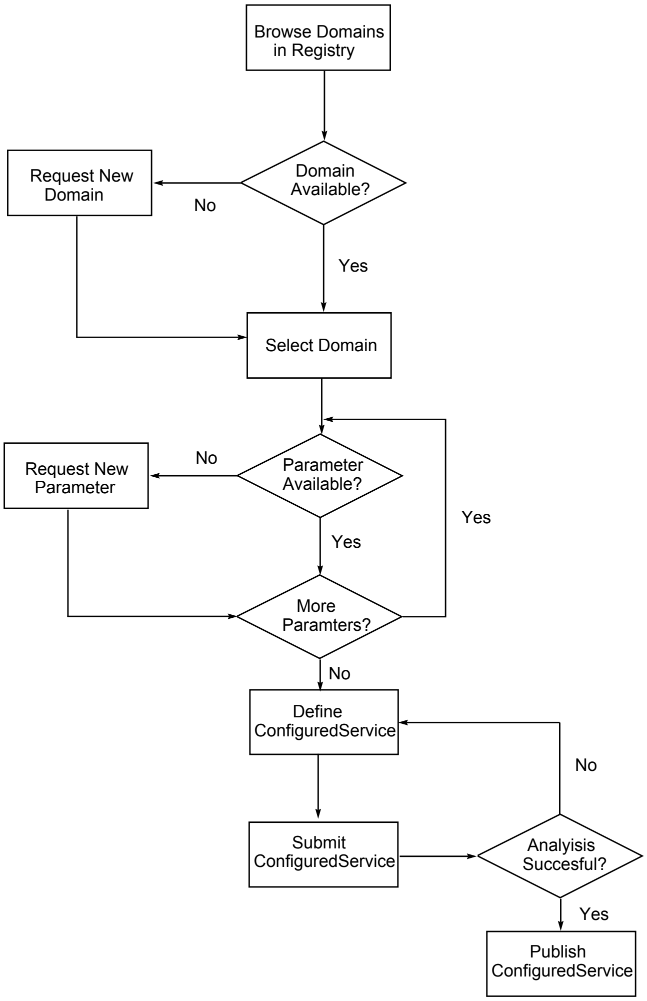


Figure 23: Service Publication Process

1. The SP selects the proper leaf domain in the SRe. This leaf domain is where the

ConfiguredService should be published. If the SP could not find an appropriate domain, a request to create a new domain is initiated by the SP. The SRe administrator will then verify that the requested new domain does not already exist and if so, a new domain is created.

2. The SP selects a proper service functionality to include in the *ConfiguredService*. If no proper service functionality is found a request to create a new service functionality is initiated by the SP. The SRe administrator will verify that the requested new service functionality does not exist and if so, a new service functionality node is created and added to the SRe.

3. The SP verifies that the *ConfiguredService* parameters are defined in the domain under the selected service functionality. The parameters include input parameters and output parameters. If any parameter does not exist, the SP can request creating a new parameter. The SRe administrator will verify that this parameter does not exist and if so, a new parameter is added to the list of parameters defined under this service.

4. The SP uses the SRL language, to be discussed in Section A.1, to specify a *ConfiguredService* and send the *ConfiguredService* SRL description and the *ConfiguredService* simplified table format to the TA. To publish a *ConfiguredService*, the SP should send the *ConfiguredService* alongside, (1) the SP information, (2) the associated domain and functionality, and (3) the SRe where the *ConfiguredService* is to be published. The domain and functionality should already be defined in the SRe.

5. The TA will perform the pre-publication analysis, as discussed in Section 4.3.1. If the analysis is successful the *ConfiguredService* is sent to the SRe for publication. Otherwise a message is sent to the SP indicating the *ConfiguredService* was rejected and the reasons for the rejection. The SP can make the necessary modifications to the *ConfiguredService* and initiate a new publication process going through the steps presented above.

6. The SP may update the information in a published *ConfiguredService*, in case that

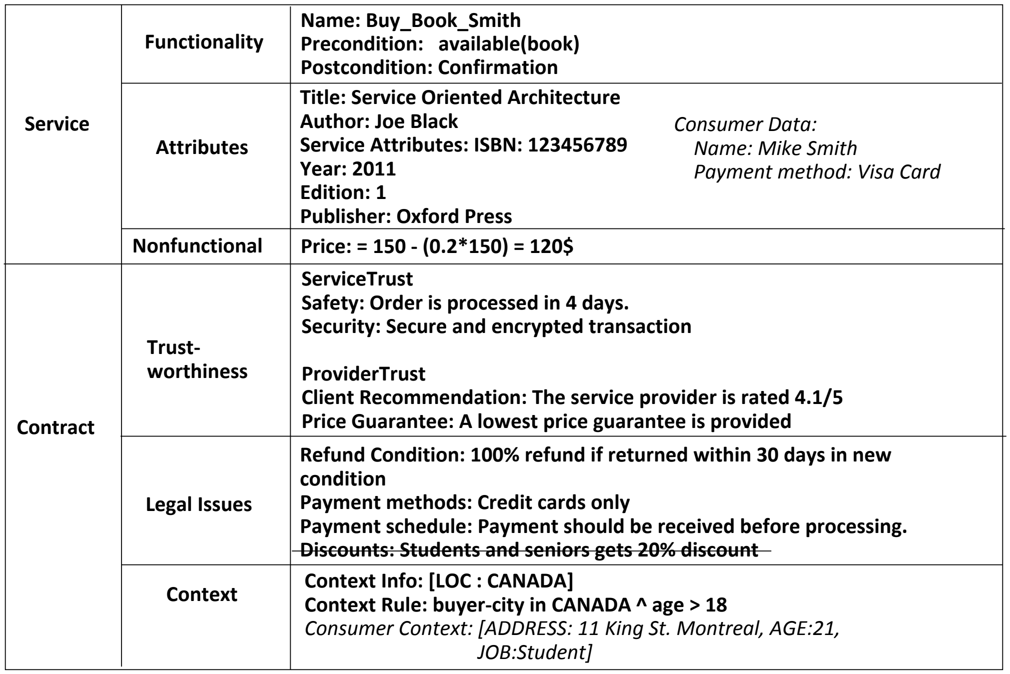
service is no longer available or a new version becomes available. If a service becomes unavailable the SP sends a message to the SRe who will remove that *ConfiguredService*. The message should include the domain, functionality, SP information and *ConfiguredService* version. In case a new version of a published service becomes available, the SP should send the new *ConfiguredService* together with the SP information, the associated domain and functionality to the TA. The domain and functionality should already be defined in the SRe.

7.3.6 Execution Unit (EU)

EU is responsible for executing plans received from the SR. A plan might contain either a single *ConfiguredService* or many *ConfiguredServices*. EU will request and receive from the SR any personal data and context information relevant to the plan. [1] For each *ConfiguredService* in the plan, the EU creates an *ExecutableService*, which is the *ConfiguredService* in which the consumer data, and consumer contextual information are added. The EU will apply the legal rules that affect the SR on the *ExecutableService*. For example, if there is a legal rule indicating a discount for a student and the requester is actually a student, the EU will update the price information accordingly. We remark that if the data and context received by the EU are not sufficient to apply all rules in the contract, the EU will demand more information from SR. Hence, the contract will be tailored to the consumer data and contextual information, and more importantly both the SP and the SR remain anonymous. Consequently, the privacy concerns of the SP are adequately dealt with in *FrSeC*.

The EU will then pass the *ExecutableService* to both SR and SP. Both parties should sign the final version. If both parties accept the current *ExecutableService* contract, the service is executed. Table 11 shows an *ExecutableService* that was generated from the Buy_Book *ConfiguredService* presented in Chapter 4.

[1] EU has the context tool-kit [Wan06] to perform the aggregation of the context information received from service requesters, providers and associated CGUs.



Service	Functionality	**Name: Buy_Book_Smith** **Precondition: available(book)** **Postcondition: Confirmation**
	Attributes	**Title: Service Oriented Architecture** **Author: Joe Black** **Service Attributes: ISBN: 123456789** **Year: 2011** **Edition: 1** **Publisher: Oxford Press** *Consumer Data:* *Name: Mike Smith* *Payment method: Visa Card*
	Nonfunctional	**Price: = 150 - (0.2*150) = 120$**
Contract	Trust-worthiness	**ServiceTrust** **Safety: Order is processed in 4 days.** **Security: Secure and encrypted transaction** **ProviderTrust** **Client Recommendation: The service provider is rated 4.1/5** **Price Guarantee: A lowest price guarantee is provided**
	Legal Issues	**Refund Condition: 100% refund if returned within 30 days in new condition** **Payment methods: Credit cards only** **Payment schedule: Payment should be received before processing.** ~~**Discounts: Students and seniors gets 20% discount**~~
	Context	**Context Info: [LOC : CANADA]** **Context Rule: buyer-city in CANADA ^ age > 18** *Consumer Context: [ADDRESS: 11 King St. Montreal, AGE:21, JOB:Student]*

Table 11: Buy_Book ExecutableService

7.3.7 Trusted Authority (TA)

The TA has the following roles.

- It provides SRs and SPs with authentication certificates (tokens) that allow them to access the SRe.
- It analyzes *ConfiguredServices* before publication and after execution.
- It formally verifies service compositions.

The TA will conduct the analyses as described in Chapter 4. It will use the UPPAAL model checking facility, discussed in Chapter 4, as a black-box for formally verifying service composition properties. Below we discuss the authentication role of TA.

Authentication Role

The TA will provide authentication certificates (tokens) to SRs and SPs. This process aims to protect the SRe.

SRs are classified, based upon the information submitted by them, in order to regulate their access to specific parts of the SRe. A SR submits legal and context information pertaining to it and requests the TA for permission to access the SRe. The legal information is defined using a policy language [And04]. It includes the identity and legal status about the SR. The context information includes information related to the service delivery location, and purpose of service request. The contextual information of the SR will decide the type of authentication certificate he will receive from the TA. As an example, a manager in a trusted financial institution might receive a different a certificate than a student whose credentials are not well established.

The TA will use the information contained in the certificate request to generate a certificate (token) that is sent to the SR, PU and the SRe. The token will specify the role assigned to the SR, which in turn will determine the extent of his access to the SRe. The token that is sent to the SR is encrypted. The SR knows that it is a token for a specific period, but will have no knowledge of the internal details of the token. This will ensure that tokens are only generated from the TA.

The SRe has the decryption key for each token and can see the internal details of every token. Figure 24 shows a simplified view of an authentication certificate. The SRe and the TA can use *Role based access control (RBAC)* methodology to control the access to the elements of the SRe. The SRe and the TA agree on a predefined set of roles. The SRe assigns access rights for each role to its elements. Upon receiving a token, the SRe decrypts it, gets the internal details including the role assigned in it. From the role specification the SRe recognizes the access rights by a simple look up in its Access Control List (ACL). The SR carrying the token is given access to the specific parts of the SRe, as defined in the ACL. After receiving a Service Query from a SP, the PU will send the certificate of the SR while sending service lookups to the SRe. Thus, only those *ConfiguredServices* that can be accessed by the access rights awarded in the certificate will become available in the *ServiceType*.

A SP may request an authentication certificate. The SP will use this certificate to browse the SRe before publishing its *ConfiguredServices*. It will follow the same steps above. It

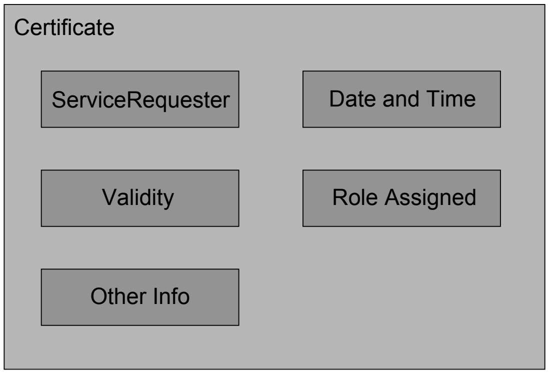


Figure 24: Authentication Certificate

can be considered a SR by itself as it is requesting a service.

7.3.8 Context Gathering Unit (CGU)

FrSeC can support multiple units to collect and transmit contextual information. In the current version of *FrSeC* three CGUs are provided. Each CGU is responsible for defining the context space, namely the set of dimensions and their associated types that are necessary to define the set of relevant context. The CGU attached to the SR will gather the contextual information that might be of interest to the SR. The SR will use that contextual information as part of its queries. The CGU will also inform the SR of any change in the contextual information, such as a change of the SR location. The SR can then choose to initialize a new service request with the updated contextual information. The CGU attached to the SP will gather the contextual information that might be of interest to the SP. The SP will use that contextual information in defining the context part of the *ConfiguredServices*. The CGU attached to the EU will gather the contextual information that might be of interest to the EU. This contextual information is related to the context in which the *ConfiguredServices* are being executed. A change in such context might violate the *ConfiguredService* context rules.

7.4 Interfaces of *FrSeC* Components

Each *FrSeC* component is seen as a black box by other components. The components interact with each other without knowing the internal details and how each component achieves its goals. In this section, we give a high-level description of the interface methods. These methods and their parameters will be refined and be made more precise at design and implementation stages. We use a lightweight formalism here to define types and interface methods. For example, sets are regarded as types, and say 'x is of type $query$ to mean that $x : Set(query)$'. The formal notation discussed in Chapter 4 can be used for typing the parameters of interface methods. The main goal of interface design is to achieve a completeness of behavior, in the sense that interfaces described below collectively achieve the goals of *FrSeC* discussed in Section 7.1.

- *Context Gathering Unit Interface:* It has one method, called *SendContext (coInfo)*. The parameter *coInfo* is of type "Context". The output of this method is a Boolean indicating the success of the sending process.

- *Service Provider Interface:* It has the following methods:

  1. *Request (sign, parms)*: This method is to be used by the EU to request service functionalities. The parameter *sign* is of type "signature" and it is the signature of the requested service. The parameter *parms* is of type "Parameter" list and it is the list of parameters sent to SP as part of the service request. The result of this method is the output of executing the service with signature *sign*.
  2. *Verify (sign)*: This method is to be used by the PNU to make sure the provider is still available and ready to provide the service. The parameter *sign* is of type "signature" and is the signature of the requested service. The result of this method is a Boolean value indicating the verification result.
  3. *Negotiation (sign, changes)*: This method is to be used by the PNU to negotiate changes to a *ConfiguredService* contract. The parameter *sign* is of type "signature" and is the signature of the service being negotiated. The parameter

changes is of type "string" and contains and required changes to the service contract. The result of this method is a new contract for the service with signature *sign*.

- *Planning Unit Interface:* It has one method, called *Query (req, qu)*. This method is to be used by the SR to query for service functionalities. The parameter *req* is of type "ServiceRequester". The parameter *qu* is of type "serviceQuery" or of type "compositionQuery" and is the query created by the SR. This method has as a precondition "the requester has an authentication certificate". The result of this method is a set of plans.

- *Service Registry Interface:* It provides the following methods:

    1. *Browse (browser, node, token)*: This method is to be used by SR or SP to browse the content of the SRe. The parameter *browser* is of type "ServiceRequester" or "ServiceProvider". The parameter *node* is of type "RegistryNode" and it is the registry node being browsed. The parameter *token* is of type "AuthenticationCertificate". The results of this method are the domain and semantic information obtained from the SRe.

    2. *Publish (pro, cs, dom, fu)*: This method is to be used by TA to publish services. The parameter *pro* is of type "ServiceProvider". The parameter "cs" is of type "*ConfiguredService*". The parameter *dom* is of type "domain" and is the domain where the published functionality belongs. The parameter *fu* is of type "functionality" and is the published functionality. The result of this method is a Boolean value indicating the success of the publication process.

    3. *Lookup (dom, fu)*: This method is to be used by the PU to search for *ConfiguredServices* in the SRe. The parameter *dom* is of type "domain" and is the requested domain. The parameter *fu* is of type "functionality" and is the required functionality. The result of this method is a set of *ConfiguredServices* that provide the functionality *fu*.

4. *add_dom (dom,node)*: This method is to be used by SPs to request to add domains to a specific node. The parameter *dom* is of type “domain” and is the domain node to be added. The parameter *node* is the node under which the domain is to be added. The precondition to this method is that *node* does not contain the added element.

5. *add_fu (fu,node)*: This method is to be used by SPs to request to add functionalities to a specific node. The parameter *fu* is of type “functionality” and is the functionality node to be added. The parameter *node* is the node under which the functionality is to be added. The precondition to this method is that *node* does not contain the added element.

6. *add_par (par,node)*: This method is to be used by SPs to request to add a parameter to a specific node. The parameter *par* is of type “parameter” and is the parameter to be added. The parameter *node* is the node under which the parameter is to be added. The precondition to this method is that *node* does not contain the added element.

- *Plan Negotiation Unit Interface:* It has the following methods:

  1. *VerifyPlan (pl)*: This method is to be used by SR to verify that all participating SP’s in a plan are still available and ready to provide the required *ConfiguredServices*. The identity of the service providers might not be known, but the address information of the *ConfiguredServiecs* are available in the plan. The parameter *pl* is of type “plan” and is the plan selected by SR. The result of this method is a Boolean value indicating the result of the verification process.

  2. *Negotiate (pl, cs, pro, changes)*: This method is to be used by SR to negotiate the contract of the participating *ConfiguredService* cs. The parameter *pl* is of type “plan” and specifies the plan to be negotiated. The parameter *cs* is of type “*ConfiguredService*” and is the service where the changes are required. The parameter *pro* is of type “Property” list. The are the properties on which there is a request for change. The parameter *changes* is of type “string” list and

specifies the required changes mapped to the property list. The result of this method is a new plan with *ConfiguredServices* that has modified contracts.

- *Execution Unit Interface:* It has one method called *Execute (req, pl, info, par)*. This method is to be used by SR to execute plans. The parameter *req* is of type "ServiceRequester". The parameter *pl* is of type "plan". The parameter *info* is of type "context" and it contains the contextual information of SR. The parameter "par" is of type "Parameter" list and it contains the input parameters provided by the SR. The result of this method is the output of executing the plan *pl*.

- *Trusted Authority Interface:* The interface has the following methods:

  1. *CertificateRequest (legalInfo, info)*: This method is to be used by SRs and SPs to request authentication certificates. The parameter *legalInfo* is of type "legalInformation". The parameter *info* is of type "context" and is the contextual information of the certificate requester. The result of this method is an authentication certificate.

  2. *Submit (cs, registry. pro, dom, fu)*: This method is to be used by SP to publish *ConfiguredServices*. The parameter *cs* is of type "*ConfiguredService*" and is the service to be published. The parameter *registry* is of type "Registry" and is the Registry where the *ConfiguredService* is to be published. The parameter *pro* is of type "ServiceProvider". The parameter *dom* is of type "domain" and is the domain where the published functionality belongs. The parameter *fu* is of type "functionality" and is the published functionality. The result of this method is either *accept* or *reject*. If the *ConfiguredService* is accepted, it will be sent to the Registry for publication and an "accept" message will be sent to the SP. If the *ConfiguredService* is rejected, a message "reject" will be sent to the SP.

  3. *Analyze (cs, pro)*: This method is to be used by SP and EU to analyze the satisfaction of a specific property in a *ConfiguredService*. The parameter *cs* is of type "*ConfiguredService*" and is the service to be analyzed. The parameter

pro is of type “Property” and is the property to be analyzed. The result of this method is a Boolean value indicating the satisfaction of the analyzed property.

4. *Verify (comp, pro)*: This method is to be used by the PU to verify service compositions. The parameter *comp* is of type “Composition” and is the service composition expression. The parameter “pro” is of type “Property” and is the property to be verified. The result of this method is a Boolean value indicating the result of verification.

5. *AnalayzeAFTER (cs, pro, exData)*: This method is to be used by the SRs to perform the after delivery analysis discussed in Section 4.3.3. The parameter *cs* is of type “*ConfiguredService*” and is the service to be analyzed. The parameter *pro* is of type “Property” and is the property to be analyzed. The parameter *exData* is of type “ExeuctionData” list and is the statistics collected while executing the *ConfiguredService cs*. The result of this method is a Boolean value indicating the satisfaction of the analyzed property.

7.5 Interaction Scenarios

From the interactions shown in Figure 18 we identify three types of scenarios. The first is the publication scenario. The second is the execution scenario. The third is the analysis scenario. These three scenarios collectively achieve the goals of *FrSeC*. Below is a discussion of each of these scenarios.

7.5.1 Publication Scenario

This scenario is performed in the following steps: (1) SP sends a request to the TA for a certificate to access SRe. (2) TA provides the certificate. (3) SP browses SRe. (4) SRe sends domain information to SP. (4a) (Optional) SP requests to add a new domain, functionality or parameter to a node in the SRe. (5) SP construct the *ConfiguredService* and sends to TA. (6) TA performs the before publication analysis discussed in Section 4.3.1

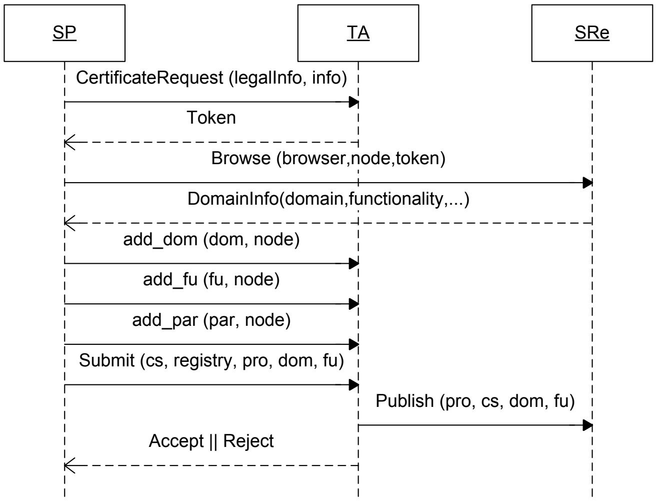

SP
TA
SRe
CertificateRequest (legalInfo, info)
Token
Browse (browser,node,token)
DomainInfo(domain,functionality,...)
add_dom (dom, node)
add_fu (fu, node)
add_par (par, node)
Submit (cs, registry, pro, dom, fu)
Publish (pro, cs, dom, fu)
Accept || Reject

Figure 25: Publication Interaction Scenario

on the *ConfiguredService*. (7) If the analysis is a success the *ConfiguredService* is sent to the SRe for publication. (8) TA sends an accept message to SP, if publication is successful, and sends a reject message otherwise. The *Message Sequence Diagram* shown in Figure 25 formalizes this scenario.

7.5.2 Execution Scenario

This scenario can be divided into a negotiation scenario and a no negotiation scenario. The no negotiation scenario is performed in the following steps: (1) SR sends a request to the TA for a certificate to access SRe. (2) TA provides the certificate. (3) SR browses SRe. (4) SRe sends domain information to SR. (5) SR constructs the query and sends it to PU. (6) PU defines and sends service lookups to SRe. (7) The service lookup result is sent from SRe to PU. (8) PU defines the query result (plan) and sends it to SR. (9) SR selects a plan and sends it to PNU. (10) PNU sends verification requests to all SP. (11) SP sends response back. (12) PNU sends verification result to SR. (13) SR sends plan to EU. (14) EU executes the plan by sending requests to SPs. (15) SP sends responses back. (16) EU sends response back to SR.

For the negotiation scenario the message sequencing differs only in steps 9, 10, 11 and 12. These are re-defined as follows: (9) SE selects a plan and sends it with the required changes to PNU. (10) PNU negotiates with SPs. (11) SP sends new *ConfiguredService*. (12) PNU sends negotiation result to SR. The *Message Sequence Diagram* in Figure 26 shows the execution scenario.

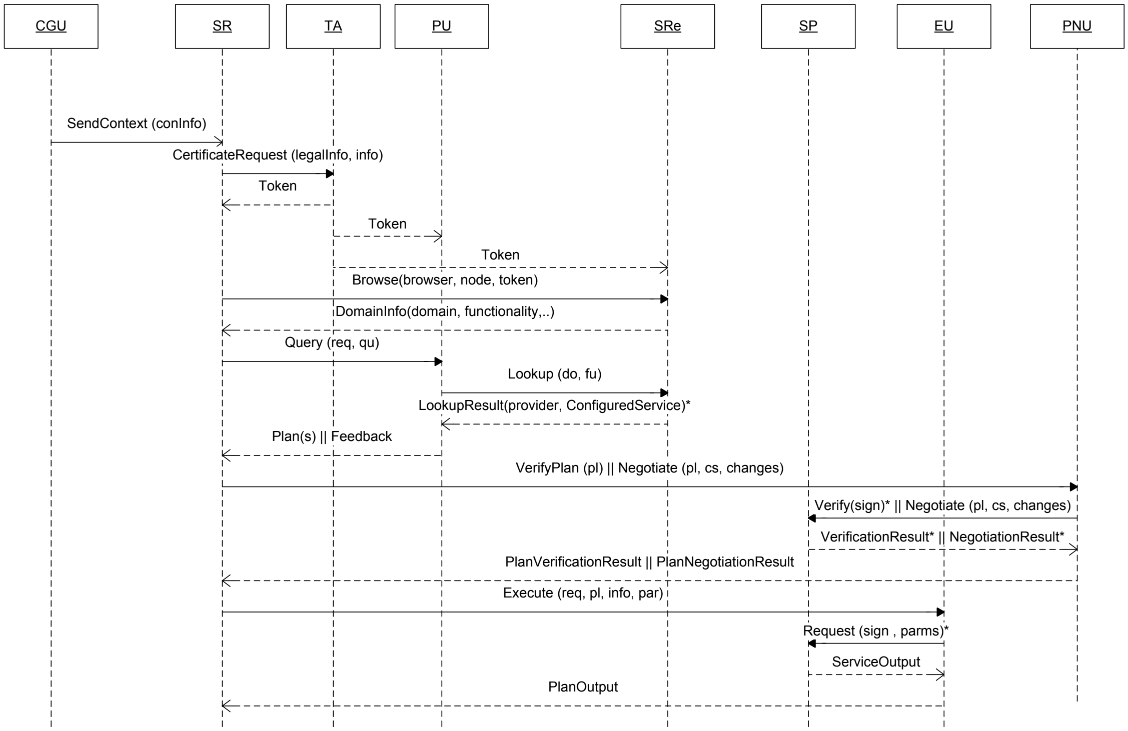

CGU
SR
TA
PU
SRe
SP
EU
PNU
SendContext (conInfo)
CertificateRequest (legalInfo, info)
Token
Token
Token
Browse(browser, node, token)
DomainInfo(domain, functionality,..)
Query (req, qu)
Lookup (do, fu)
LookupResult(provider, ConfiguredService)*
Plan(s) || Feedback
VerifyPlan (pl) || Negotiate (pl, cs, changes)
Verify(sign)* || Negotiate (pl, cs, changes)
VerificationResult* || NegotiationResult*
PlanVerificationResult || PlanNegotiationResult
Execute (req, pl, info, par)
Request (sign , parms)*
ServiceOutput
PlanOutput

Figure 26: Execution Interaction Scenario

7.5.3 Analysis Scenario

This scenario represents the analysis performed by the TA. The TA performs the analysis on behalf of the SR, SP, PU and EU. The interactions can be formalized using the *Message Sequence Diagram* shown in Figure 27.

It is easy to verify that collectively the interface methods discussed in the previous section are sufficient to describe the *FrSeC* interactions shown in Figure 18. The three scenarios scenarios discussed in this section completely cover the set of interactions in Figure 18.

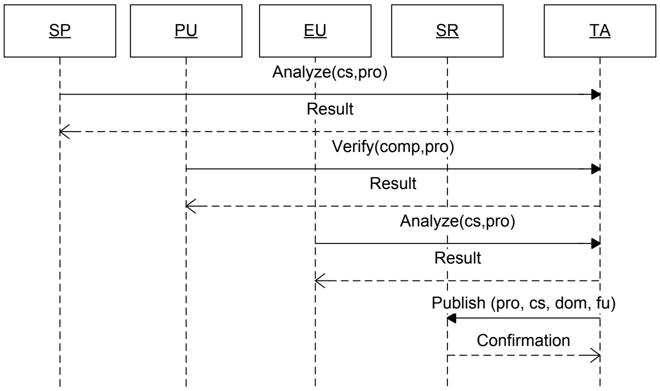


Figure 27: Analysis Interaction Scenario

This observation justifies our claim that the interface methods achieve 'communication completeness' of *FrSeC*.

7.6 *FrSeC* Adaptability

One of the main features of *FrSeC* is its ability to adapt to situations that trigger a need for a rediscovery or re-ranking process. Below is a discussion of the most important triggers and how they are handled in *FrSeC*.

Context change: The discovery process uses the contextual information of the SR at service discovery time. But during service execution, the contextual information of the SR might have changed. As a consequence, the contextual rules of the discovered service(s) might be violated, other services may be more suitable. In order to deal with the dynamic change in context we introduce an adaptable discovery mechanism. In *FrSeC*, this mechanism includes the following steps:

1. CGU senses the new context information and informs SR.
2. SR generates a new query with the new context information.
3. The context change may result in a change to the security level. So SR contacts TA with the new context information.

4. TA sends a new authentication certificate to SR.

5. SR sends the new query to the PU which will initiate a new discovery process.

6. PU will send a new plan with the set of the new ranked *ConfiguredServices*.

7. EU will migrate from the old *ConfiguredService* to the new *ConfiguredService*.

Failure in service availability: During service execution, the executing service might fail or become unavailable. For example, the wireless router might fail. *FrSeC* is designed to adapt to service failures. In our design, PU uses *ServiceType*, and not specific *ConfiguredServices* when defining query result. A *ServiceType* contains ordered *ConfiguredServices* that can meet the requirements of a specific query. During run time, if a *ConfiguredService* fails or becomes unavailable, the EU will select the next *ConfiguredService* in the *ServiceType*. The worst case is that an equivalence class has only one *ConfiguredService* and it fails. The feedback loop in *FrSeC* will restart the service discovery process in this case.

New alternative services: Service executions may be performed over days, or even months. But service selection and binding are usually performed only the first time the requester uses the service. This might not be practical because new services might be available during this long execution time. The new alternative services might be new services or old services with new modified contracts. A new contract might include a lower price or a better quality. For example, a wireless provider with cheaper price and same quality guarantees might become available. In order to adapt to new alternative services during run time, SR registers with PU. This registration will guarantee that PU will inform SR in case a new *ConfiguredService* that provides the same functionality becomes available. Thus, SR can initiate a new discovery process.

New contract rules: Contracts bound to *ConfiguredServices* may be either *strict* or *flexible*. In a strict contract, the life-time of contract is made explicit. Providers and requesters are bound by this timeline. In a flexible contract, there is no life-time specification, which allows providers to change the contract terms at any point of time. For example, the SP might increase the price of his wireless Internet connection. Providers might not be aware of the identity of their clients. This design decision was made to enshrine privacy issues. In *FrSeC*, providers inform EU of changes to service contract. At the time of service delivery, EU informs SR of changes to the contract and delivers the service only upon receiving the acceptance of new contract terms from SR. In order not to deny service, requesters are allowed to initiate a rediscovery process in accordance with the new contract terms.

New requester requirements: Some service executions might be too long and during this service time the requirements of the requester might change. To deal with new requirements, the requester has the choice of a rediscovery process or a re-ranking process. In the rediscovery process the requester will define a new query and go through all steps of service discovery. In a re-ranking process, the requester will ask PU to re-rank the *ConfiguredServices* in the *ServiceType* taking into consideration the new assigned weights to the elements in the modified query.

7.7 Case Study - Auto Roadside Emergency Service

This section will use the auto road assistance example presented in Section 4.5 to illustrate the operation of *FrSeC*. Here, we are focusing on single service request and response, and do not consider compositions. We will assume that the requests are done sequentially and not simultaneously. First a request for repair shop is performed, and then the driver makes an appointment with this repair shop. Second, a request for a tow truck is performed, and then the driver calls the tow truck company. Finally, a request for a car rental is performed, and the driver calls the car rental company. We will focus here on illustrating *FrSeC* operations in fulfilling requests to a repair shop. A similar approach can be followed for tow truck and car rental services.

Service Publication: Repair shops SPs communicate with the TA to obtain the authentication certificates that enables them to access the SRe. The SPs search the SRe for the appropriate domain until they find the *Repair shop* domain. Under this domain they search for the appropriate functionality which is in this case *Reserve*. Then, they will verify that their parameters are defined under the *Reserve* functionality. Next, SPs will publish the *ConfiguredServices* by submitting them to the TA. Figure 28 shows the part of SRe for repair shop. We have 3 SPs, and they provide the 5 *ConfiguredServices* presented in Figure 29. All *ConfiguredServices* have the same input parameters which are *(CarBroken:bool)*, *(Deposite:double)* and *(CarType:string)*. They also have the same output parameters which are *(HadAppointment:bool)* and *(NumOfHours:int)*.

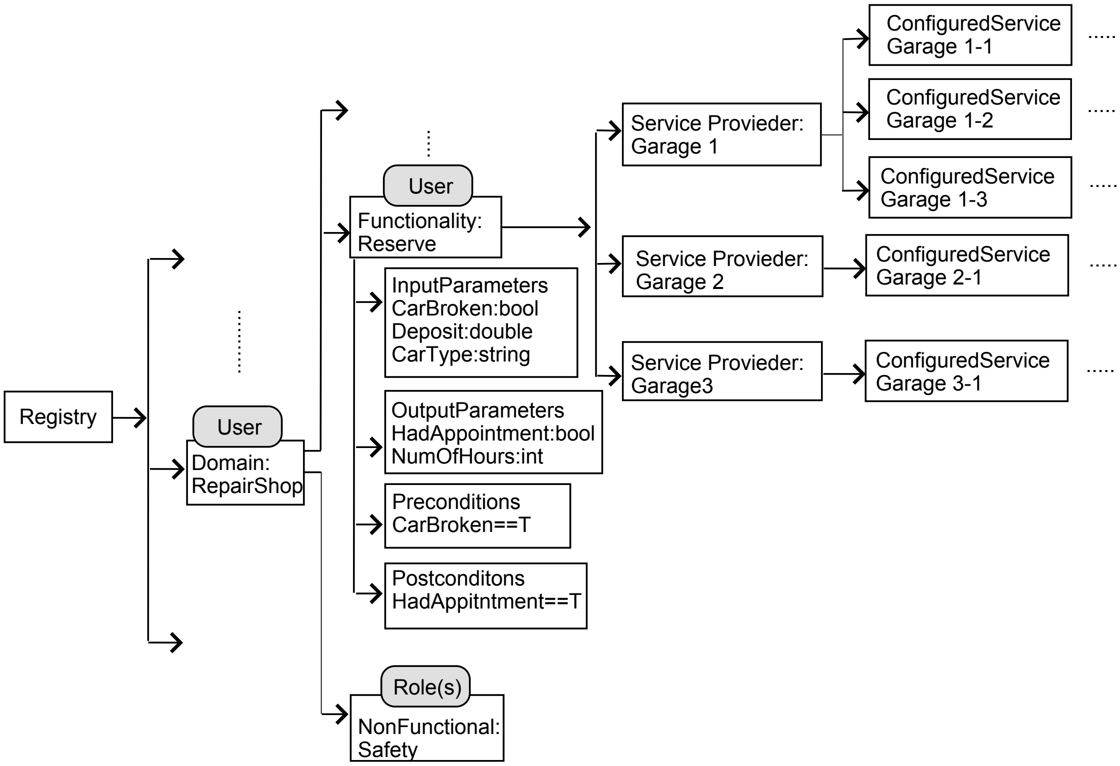
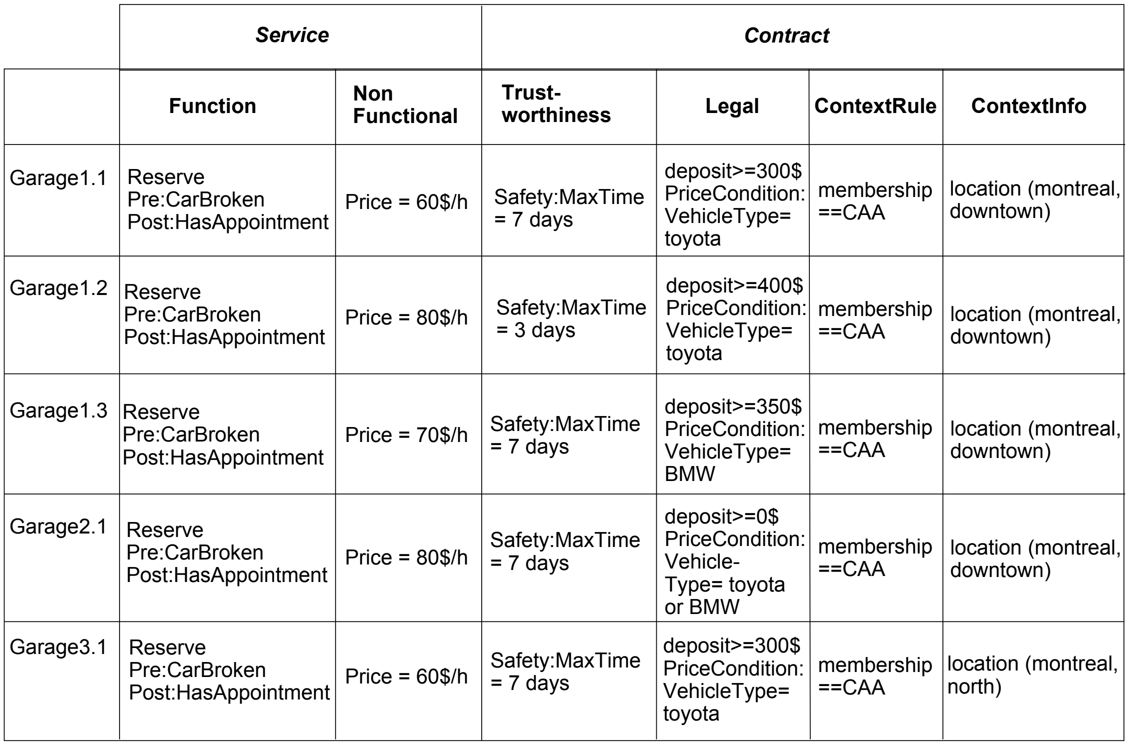


Figure 28: Service Registry Structure for the Case Study

Querying: The SR, in this case the *Vehicle*, accesses the SRe searching for the appropriate domain and functionality. But before doing that, an authentication certificate from the TA is needed. Figure 30 shows the certificate sent to the SR. The SR will use this certificate to access the SRe to find the domain and functionality, in this case *Repair shop* and *Reserve*. The SR will then access the functionality parameters and will use them in defining the Service Query. Figure 31 shows three traditional exact-type Service Queries. Each Service



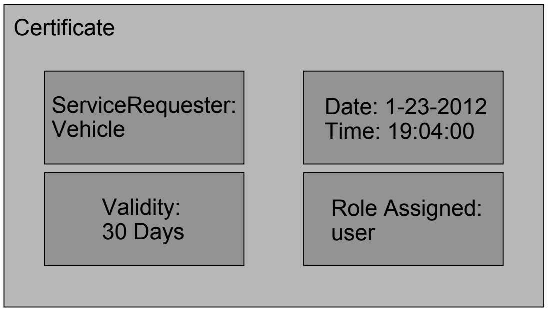
	Service		***Contract***			
	Function	**Non Functional**	**Trust-worthiness**	**Legal**	**ContextRule**	**ContextInfo**
Garage1.1	Reserve Pre:CarBroken Post:HasAppointment	Price = 60$/h	Safety:MaxTime = 7 days	deposit>=300$ PriceCondition: VehicleType= toyota	membership ==CAA	location (montreal, downtown)
Garage1.2	Reserve Pre:CarBroken Post:HasAppointment	Price = 80$/h	Safety:MaxTime = 3 days	deposit>=400$ PriceCondition: VehicleType= toyota	membership ==CAA	location (montreal, downtown)
Garage1.3	Reserve Pre:CarBroken Post:HasAppointment	Price = 70$/h	Safety:MaxTime = 7 days	deposit>=350$ PriceCondition: VehicleType= BMW	membership ==CAA	location (montreal, downtown)
Garage2.1	Reserve Pre:CarBroken Post:HasAppointment	Price = 80$/h	Safety:MaxTime = 7 days	deposit>=0$ PriceCondition: VehicleType= toyota or BMW	membership ==CAA	location (montreal, downtown)
Garage3.1	Reserve Pre:CarBroken Post:HasAppointment	Price = 60$/h	Safety:MaxTime = 7 days	deposit>=300$ PriceCondition: VehicleType= toyota	membership ==CAA	location (montreal, north)

Figure 29: Available *ConfiguredServices*

Query is sent to the PU. The PU will then send service lookups to the SRe. The lookups result will be matched with the Service Query requirements by the PU. Finally, the matching result is send to the SR. Figure 32 shows the results returned by the PU.

The vehicle will then send a request for a reservation to the selected repair shop. In this example, we are not concerned about the internal details of the vehicle. We assume it is a composite service and we deal with it as a black box. The internal services, such as the GPS location service and the engine sensor service are not of concern for us at this stage. We deal with the Vehicle as a single service.

Figure 33 shows the interaction activity performed in the *FrSeC* framework to find the most appropriate repair shop and request an appointment.



Certificate

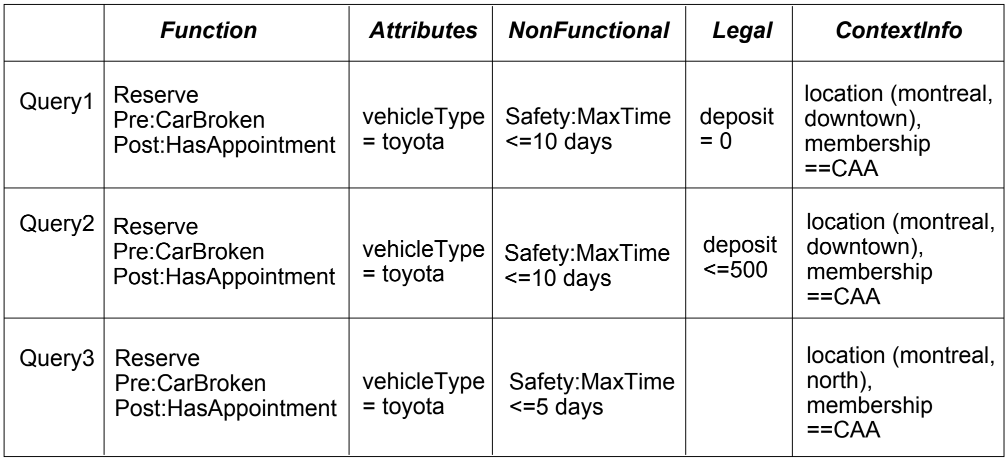
ServiceRequester: Vehicle

Date: 1-23-2012
Time: 19:04:00

Validity: 30 Days

Role Assigned: user

Figure 30: Vehicle Authentication Certificate

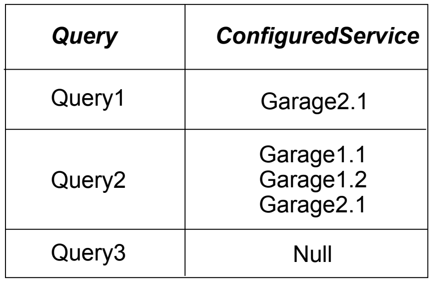


	Function	***Attributes***	***NonFunctional***	***Legal***	***ContextInfo***
Query1	Reserve Pre:CarBroken Post:HasAppointment	vehicleType = toyota	Safety:MaxTime <=10 days	deposit = 0	location (montreal, downtown), membership ==CAA
Query2	Reserve Pre:CarBroken Post:HasAppointment	vehicleType = toyota	Safety:MaxTime <=10 days	deposit <=500	location (montreal, downtown), membership ==CAA
Query3	Reserve Pre:CarBroken Post:HasAppointment	vehicleType = toyota	Safety:MaxTime <=5 days		location (montreal, north), membership ==CAA

Figure 31: Repair Shop Requests

7.8 Summary

This chapter has presented *FrSeC*, the formal framework for the provision of context-dependent services. The components of *FrSeC*, their roles, interfaces, and interaction scenarios have been described in a variety of notations, ranging from formal to informal. The *FrSeC* description is sufficiently comprehensive that it should be possible to derive a detailed component-based design, which will ultimately lead to a faithful implementation of it.



Query	***ConfiguredService***
Query1	Garage2.1
Query2	Garage1.1 Garage1.2 Garage2.1
Query3	Null

Figure 32: Planning Unit Results

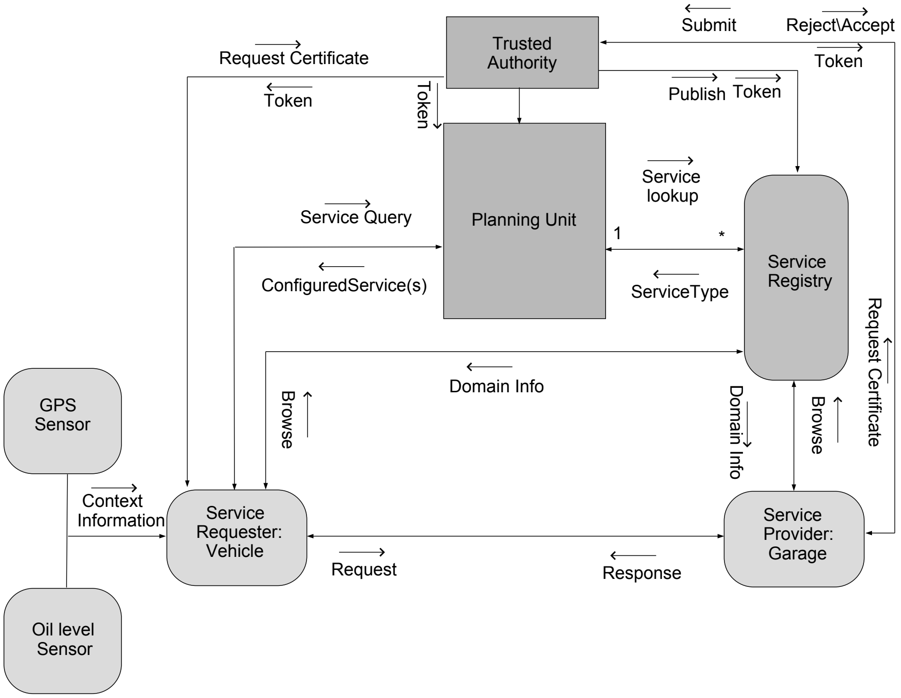


Figure 33: *FrSeC* Interaction for the Case Study

Chapter 8

Dynamic Composition

Static service composition is driven by provider's needs and hence can be defined independently from the service provision framework. On the other hand, dynamic service composition is driven by requester's needs and hence is closely related to the service provision framework. This chapter discuses the three types of dynamic service composition *template-based*, *semi-automatic* and *automatic* introduced by *FrSeC*.

8.1 Template-based Composition

Service requesters browse the Service Registry looking for functionalities that meet their requirements. In many cases, the required service functionality is so complex that it may not be directly available in the Service Registry. In such cases, the service requester is forced to find multiple functionalities from the Registry that collectively might be sufficient to meet his requirements. To facilitate the service requester in formulating such complex queries we are introducing the template-based composition queries. A composition template suggests how the services that match the query might be composed. The service requester formulates a template query and passes it to the Planning Unit, who executes it according to the composition semantics of services that match the template query. The protocol, shown in Figure 34, is as follows:

1. The service requester browses the Service Registry and does not find a match to its

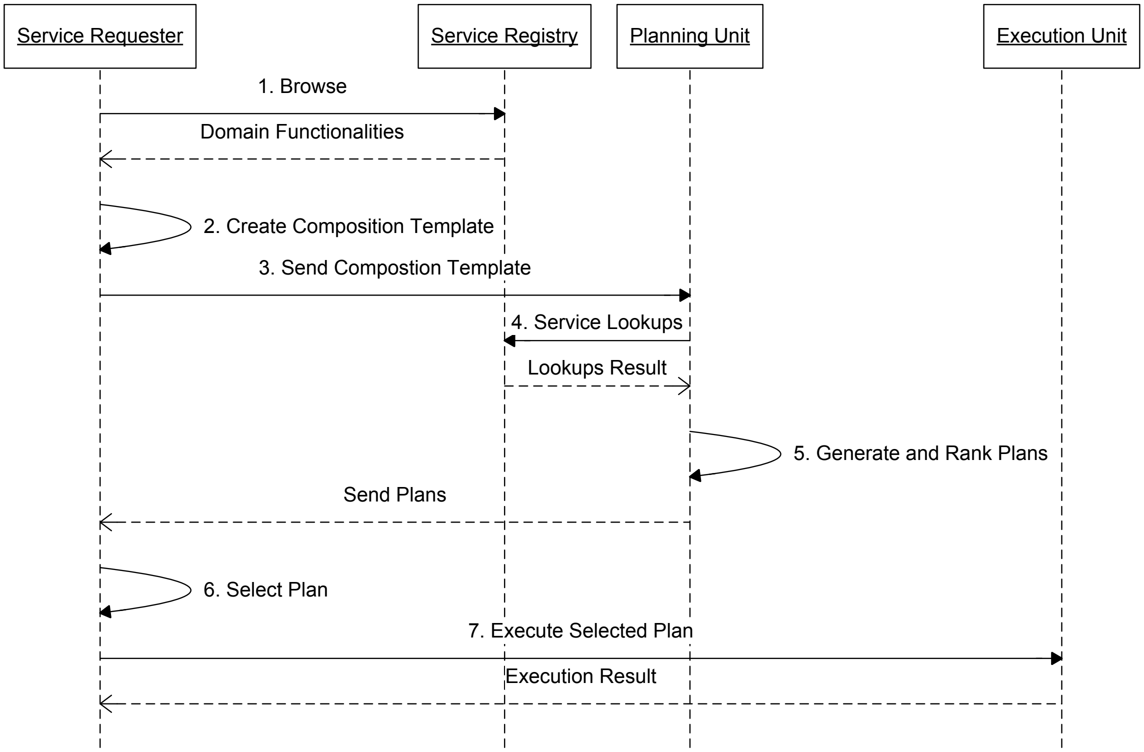


Figure 34: Template-based Composition Protocol

required functionality but it rather finds several partial matches to its requirements.

2. The service requester uses the partial matches to create a composition template that meets its requirements.

3. The service requester sends the composition template to the Planning Unit.

4. The Planning Unit consults the Registry to find matches for the requirements defined in the composition plan.

5. The Planning Unit may generate multiple composition plans, and ranks them according to the preferences assigned by the service requester. The Planning Unit will then send the ranked plans to the service requester.

6. The service requester will select the best plan suited for him.

7. The service requester sends the composition plan to the Execution Unit for execution.

8.1.1 Template-based Composition Query

A template-based composition query (TCQ) is formed very similar to the way a service expression is formed. We have chosen the query composition constructs that correspond in a one-to-one manner with static service composition operators (discussed in Section 5.1), for two reasons. One reason is that these operators seem sufficient to express a large number of complex queries. The second reason is that we want the query processing activity to be simplified without too much overhead to the planner.

The composition query operators are $\circledast$ (for sequential composition), $\ltimes$ (for parallel composition), $\circledcirc$ (for priority composition), $\oplus$ (for no order composition), † (for nondeterministic choice composition), $\odot$ (for conditional choice composition), and $\oslash$ (for iterative composition). Below we explain their semantics.

- **Sequential Query Composition** Let X and Y denote two traditional-style queries. The sequential composition of X and Y is a query Z, written $X \circledast Y$. Let R_X and R_Y denote the set of *ConfiguredServices* that respectively match the queries X and Y. The intended response to query Z is the set $R_Z = \{s_X \gg s_Y \mid s_X \in R_X, s_Y \in R_Y\}$. That is, the intention of the service requester is that a service that matches query X is to be executed first, and immediately following that a service that matches the query Y should be executed. This wish of the service requester is respected by the service provider by applying sequential composition construct to compose services that match the input queries. Clearly, sequential query composition can be generalized to involve k, $k > 2$ queries. Corresponding to a generalized sequential query composition, the set of sequential compositions of their matching services can be produced.

- **Parallel Query Composition** Let X and Y denote two traditional-style queries. The parallel composition of X and Y is a query Z, written $X \ltimes Y$. Let R_X and R_Y

denote the set of *ConfiguredServices* that respectively match the queries X and Y. The intended response to query Z is the set $R_Z = \{s_X || s_Y \mid s_X \in R_X, s_Y \in R_Y\}$. That is, the intention of the service requester is that a service that matches query X and a service that matches the query Y are to be executed simultaneously. The service provider fulfills this demand. Clearly, parallel query composition can be generalized to involve k, $k > 2$ queries. Corresponding to a generalized parallel query composition, the set of parallel compositions of their matching services can be produced.

- **Priority Query Composition** Let X and Y denote two traditional-style queries. The priority composition of X and Y is a query Z, written $X \circledcirc Y$. Let R_X and R_Y denote the set of *ConfiguredServices* that respectively match the queries X and Y. The intended response to query Z is the set $R_Z = \{s_X \prec s_Y \mid s_X \in R_X, s_Y \in R_Y\}$. That is, the intention of the service requester is to request the execution of a service that matches query X, if it succeeds. Otherwise, the service requester is willing to accept a service matches query Y. The service provider fulfills this request. Clearly, priority query composition can be generalized to involve k, $k > 2$ queries. Corresponding to a generalized priority query composition, the set of priority compositions of their matching services can be produced.

- **No Order Query Composition** Let X and Y denote two traditional-style queries. The no order composition of X and Y is a query Z, written $X \oplus Y$. Let R_X and R_Y denote the set of *ConfiguredServices* that respectively match the queries X and Y. The intended response to query Z is the set $R_Z = \{s_X \Diamond s_Y \mid s_X \in R_X, s_Y \in R_Y\}$. That is, the intention of the service requester is to receive the services that match query X and query Y in any order. Either one of them can be started first. This is fulfilled by the service provider. Clearly, no order query composition can be generalized to involve k, $k > 2$ queries. Corresponding to a generalized no order query composition, the set of no order compositions of their matching services can be produced.

- **Nondeterministic Choice Query Composition** Let X and Y denote two traditional-style queries. The nondeterministic choice composition of X and Y is a query Z, written $X \dagger Y$. Let R_X and R_Y denote the set of *ConfiguredServices* that respectively match the queries X and Y. The intended response to query Z is the set $R_Z = \{s_X \wr s_Y \mid s_X \in R_X, s_Y \in R_Y\}$. That is, the intention of the service requester is to randomly accept a service that matches query X or a service that matches the query Y. This is fulfilled by the service provider. Clearly, nondeterministic choice query composition can be generalized to involve k, $k > 2$ queries. Corresponding to a generalized nondeterministic choice query composition, the set of nondeterministic choice compositions of their matching services can be produced.

- **Conditional Choice Query Composition** Let X and Y denote two traditional-style queries. The conditional choice composition of X and Y is a query Z, written $X \odot_c Y$. Let R_X and R_Y denote the set of *ConfiguredServices* that respectively match the queries X and Y. As a response to query Z the service provider computes the set $R_Z = \{s_X \rhd_c s_Y \mid s_X \in R_X, s_Y \in R_Y\}$, and provides one or more from the set to the service requester. That is, the intention of the service requester is that if condition c is true then a service that matches query X is to be executed, otherwise a service that matches the query Y is executed.

- **Iteration Query Composition** Let X denotes a traditional-style query. The iteration composition of X is a query Z, written $X_{\oslash_c}$. Let R_X denotes the set of *ConfiguredServices* that match the query X. The intended response to query Z is the set $R_Z = \{s_{X \circ_c} \mid s_X \in R_X\}$. That is, the intention of the service requester is to receive a service that matches query X as long as it can be executed to meet the condition c.

All query operators have equal priority. From the query composition semantics it is clear that (1) every query composition, as suggested above, has a unique service interpretation, and (2) a query expression can be uniquely transformed into a service expression. As an example, corresponding to the query expression $X \dagger Y \oslash_c$ there exists a set of service expressions given by $\{s_X \wr s_Y \circ_c \mid s_X \in R_X, s_Y \in R_Y\}$.

Definition 17 *A template-based query composition is an expression*

$$q_1 \star q_2 \star q_3 \star .. \star q_i,$$

where $\star \in \{\circledast, \ltimes, \circledcirc, \oplus, \dagger, \odot, \oslash\}$ *and* q_i *is a simple traditional style query. All query composition operators have the same precedence. A query expression is evaluated from left to right.*

8.1.2 Ranking of Candidate Compositions

Each template-based composition query involves many traditional style queries. For each traditional style query many *ConfiguredServices* might be selected by the Planning Unit according to the matching algorithms discussed in Section 7.3.3. Therefore, there are many possible composition plans that match a template-based composition query. It is necessary for the Planning Unit to rank the candidate composition plans in order to enable the service requester to make an intelligent selection.

As discussed is Section 7.3.3, each *ConfiguredService* that is matched to a traditional style query will have a ranking value. This ranking value is calculated by the Planning Unit using the weights assigned by the service requester to each property in the traditional style query. In template-based composition query, the service request can assign a weight to each individual traditional style query and to each property inside the traditional style query. These values will all be used in ranking the composition plans.

Let $SQ_1, \ldots, SQ_n$ denote the traditional style service queries in a composition query, and $w_1, \ldots, w_n$ denote the weights assigned by the service requester to the traditional queries. Let $CS_1, .., CS_n$ denote the *ConfiguredServices* in the composition plan, and PM_{ij} be the percentage match of SQ_i with candidate *ConfiguredService* CS_j. The percentage match PM_{ij} is calculated using Equation 5, where CSR_{ij} is the ranking value of *ConfiguredService* j with respects to traditional query i as calculated in Section 7.3.3. If MR_i is the maximum ranking value for any *ConfiguredService* with respect to query i then

	Case 1	*Case 2*
SQ_1	$CS_1 = 0.90$	$CS_1 = 0.80$
SQ_2	$CS_3 = 0.85$	$CS_4 = 0.80$
SQ_3	$CS_5 = 0.95$	$CS_6 = 0.90$

Table 12: Percentage Match of Candidate ConfiguredServices

$$PM_{ij} = \frac{CSR_{ij}}{MR_i} \tag{5}$$

The values w_i are provided by the service requester. Hence the ranking value RV for a candidate composition plan can be calculated according to Equation 6.

$$RV = (w_1 * PM_{11}) + (w_2 * PM_{22}) + \cdots + (w_n * PM_{nn}) \tag{6}$$

Example 15 *In this example, we illustrate the ranking of service compositions as performed by the Planning Unit. Let $SQ_1 \circledast SQ_2 \circledast SQ_3$ be the composition query template created by a service requester. Assume that for query SQ_1 the two candidate matching* ConfiguredServices *are CS_1 and CS_2, for query SQ_2 the two candidate matching* ConfiguredServices *are CS_3 and CS_4, and for SQ_3 the two candidate matching* ConfiguredServices *are CS_5 and CS_6. Hence, we have 8 possible composition plans. We also assume that the percentage match of each* ConfiguredService *is shown in Table 12 for all service queries.*

The service requester assigns a priority to each service query. Table 13 illustrates the ranking that resulted from assigning two different set of priorities to the service query. In the first case, the service requester assigns weight 4 to SQ_1, SQ_2 and SQ_3. In the second case, the service requester assigns weight 1 to SQ_1, weight 3 to SQ_2 and weight 1 to SQ_3.

Table 13 shows the ranking value for each candidate composition plan and the ranking of each plan. To illustrate the method of calculating each ranking value, let's take for example the composition $CS_1 \gg CS_3 \gg CS_5$. Using the first set of weights, the ranking

	Case 1 (Rank)	*Case 2 (Rank)*
$CS_1 \gg CS_3 \gg CS_5$	*10.8 (1)*	*4.40 (1)*
$CS_1 \gg CS_3 \gg CS_6$	*10.6 (2)*	*4.35 (2)*
$CS_1 \gg CS_4 \gg CS_5$	*10.6 (2)*	*4.25 (4)*
$CS_1 \gg CS_4 \gg CS_6$	*10.4 (3)*	*4.20 (5)*
$CS_2 \gg CS_3 \gg CS_5$	*10.4 (3)*	*4.30 (3)*
$CS_2 \gg CS_3 \gg CS_6$	*10.2 (4)*	*4.25 (4)*
$CS_2 \gg CS_4 \gg CS_5$	*10.2 (4)*	*4.15 (6)*
$CS_2 \gg CS_4 \gg CS_6$	*10.0 (5)*	*4.10 (7)*

Table 13: Ranking Candidate Compositions

value is calculated as follows:

$$RankingValue = (0.9 * 4) + (0.85 * 4) + (0.95 * 4) = 10.8$$

For the second set of weights, the ranking value is calculated as follows:

$$RankingValue = (0.9 * 1) + (0.85 * 3) + (0.95 * 1) = 4.4$$

This example illustrates the ranking algorithm used by the Planning Unit to rank the different composition plans. It illustrates that for each set of weights assigned by the service requester a different ranking will result.

8.2 Semi-automatic Composition

In template-based composition, the assumption is that a service requester is fully aware of all his requirements. Hence, it is easy for him to define his service query template. In many cases, the service requester is not clear about all requirements. The requirements might be building gradually. A simple travel planning example can illustrate this issue. In this example a service requester is planning to buy an airplane ticket to New York and reserve a hotel close to the arrival airport. But New York has three international airports JFK, Newark and LaGuardia. Therefore, the service requester is not able to define the query for the hotel without first buying the air ticket. Hence, the service requester should be able to first query for the air tickets, receive all candidate *ConfiguredServices*, select a *ConfiguredService* and then query for the hotel taking into consideration the selected air ticket and the arriving airport.

In such cases, the service requester should be able to query for functionalities and matching *ConfiguredServices* in a gradual manner. Hence, we introduce semi-automatic service composition. In semi-automatic composition, the service requester incrementally queries for functionalities that match his most recent requirements. At each step, the service requester will query for one functionality using a traditional query (discussed in Section 7.3.2). The query is sent to the Planning Unit which responds with a set of candidate *ConfiguredServices*. The service requester then selects a *ConfiguredService* from the candidates.

The main difference between semi-automatic composition and multiple regular traditional queries for single *ConfiguredServices* is that in multiple traditional queries the process is stateless. In the sense that the Planning Unit deals with each traditional query separately. The Planning Unit does not keep information about previous selections and is actually not aware of what *ConfiguredService* was selected. On the other hand, in semi-automatic composition the process is stateful. The Planning Unit creates a session for each semi-automatic composition process. In this session, the requests and selections are saved. This will enable the Planning Unit to compose all selected *ConfiguredServices* and create a composite service that meets the requirements of the service requester.

The semi-automatic service composition process can be summarized in the following steps, shown in Figure 35:

1. The service requester browses the Service Registry for all required functionality.

2. The service requester will create a semi-automatic initialization composition query.

3. The service requester will send the initialization query containing a traditional query for the first functionality to the Planning Unit.

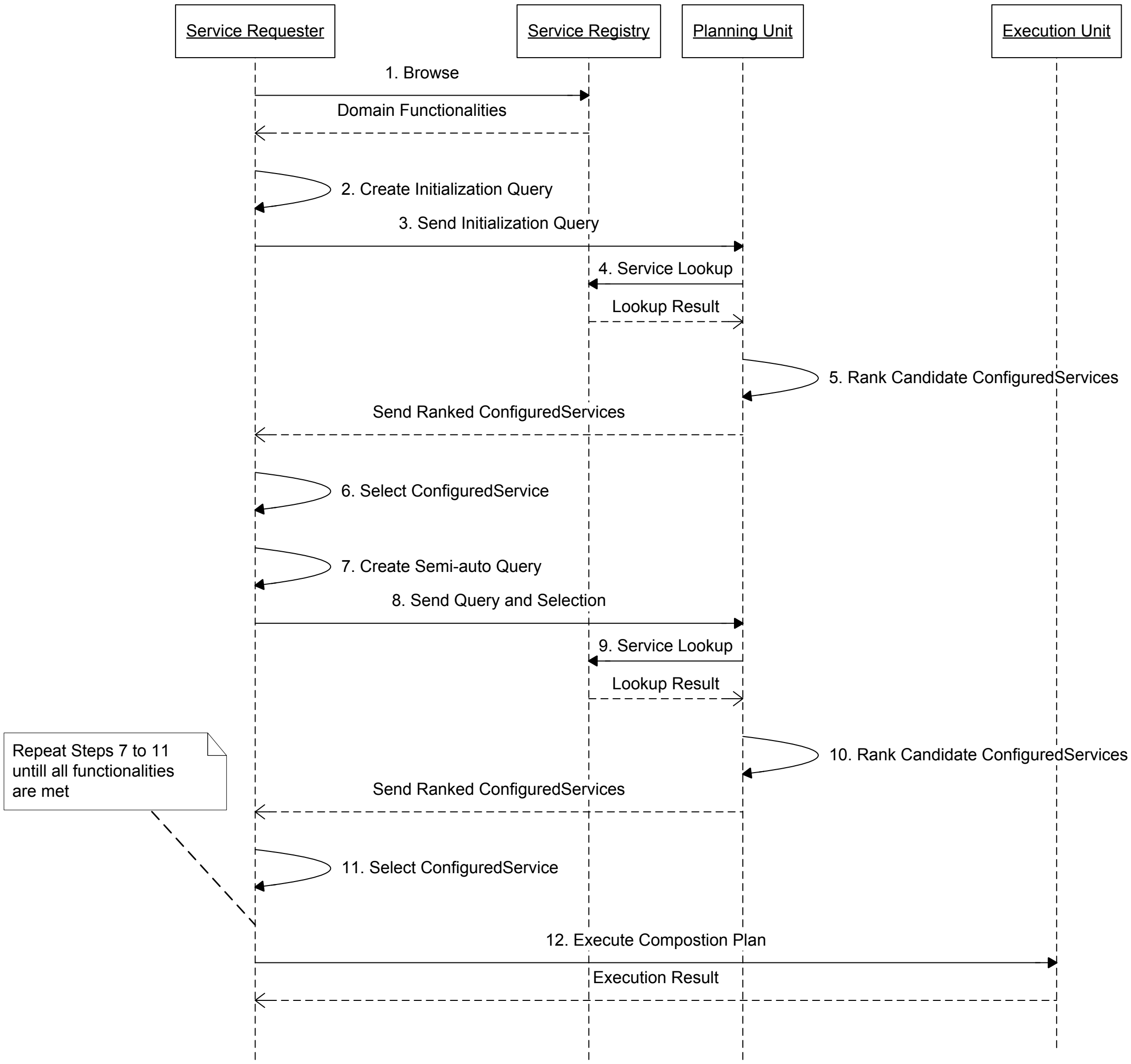

Service Requester
Service Registry
Planning Unit
Execution Unit
1. Browse
Domain Functionalities
2. Create Initialization Query
3. Send Initialization Query
4. Service Lookup
Lookup Result
5. Rank Candidate ConfiguredServices
Send Ranked ConfiguredServices
6. Select ConfiguredService
7. Create Semi-auto Query
8. Send Query and Selection
9. Service Lookup
Lookup Result
Repeat Steps 7 to 11 untill all functionalities are met
10. Rank Candidate ConfiguredServices
Send Ranked ConfiguredServices
11. Select ConfiguredService
12. Execute Compostion Plan
Execution Result

Figure 35: Semi-automatic Composition Protocol

4. The Planning Unit consults the Registry to find matches for the requirements defined in the initialization query.

5. The Planning Unit will match the lookup result with the service requester requirements and will send the matched and ranked *ConfiguredServices* to the service requester.

6. The service requester will then select a specific *ConfiguredService* from the list of candidate *ConfiguredServices*.

7. The service requester will then create a new semi-automatic query containing a traditional query for the second required functionality.

8. The service requester will then send the semi-automatic query to the Planning Unit.

9. The Planning Unit consults the Registry to find matches for the requirements defined in the semi-automatic query.

10. The Planning Unit will match the lookup result with the service requester requirements taking into consideration previous selections and will send the matched and ranked *ConfiguredServices* to the service requester.

11. The service requester will then select a specific *ConfiguredService* from the list of candidate *ConfiguredServices*.

    Steps 7 to 11 are repeated until all required functionalities are met.

12. The service requester will send the final composition plan received from the Planning Unit to the Execution Unit for execution.

8.2.1 Semi-automatic Composition Query

The semi-automatic composition process is performed in many steps. At each step a query is sent by the service requester to the Planning Unit. From the semi-automatic composition process we can acknowledge three stages of querying. The first stage is to form an initialization query. The second stage is to form a semi-automatic query. The final stage is to form the ending query.

The initialization query has two main goals. The first is to inform the Planning Unit that a semi-automatic composition process is starting, which will tell the Planning Unit to create a new session. The second goal is to define the first required functionality. The initialization contains two main parts. The first part defines a unique name for the semi-automatic composition process. The second part is a traditional style query defining the first required functionality and associated nonfunctional, trustworthiness, legal and context information.

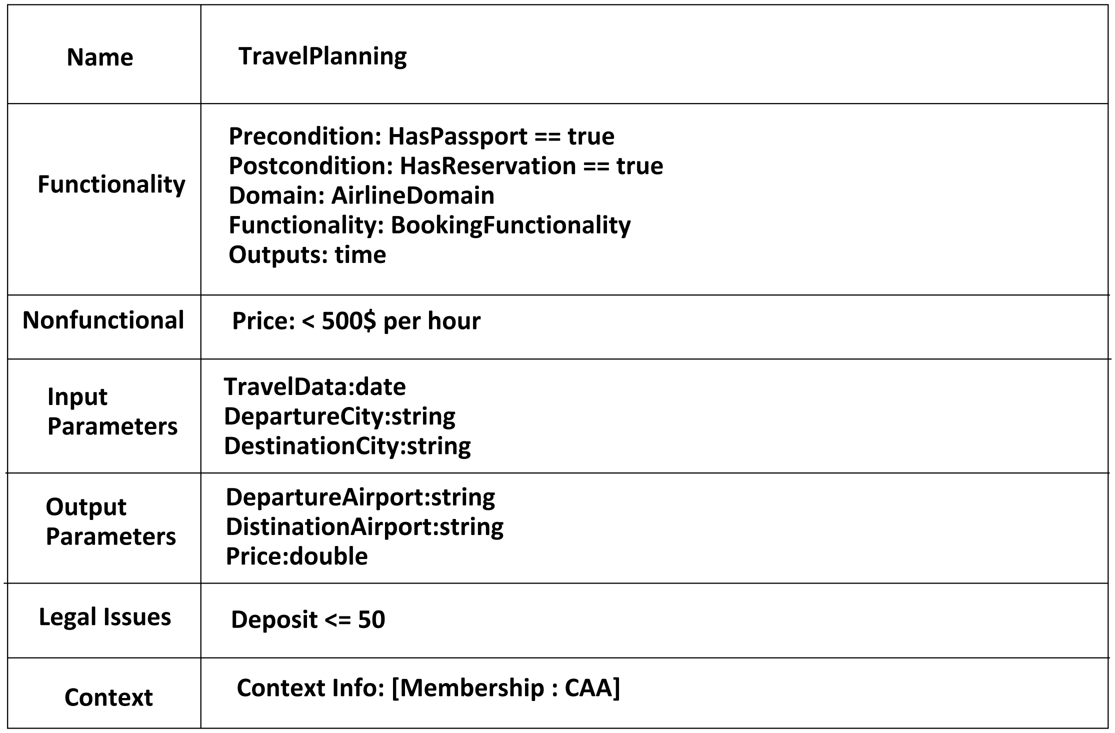
Definition 18 *The semi-automatic initialization query $q_{(sa)_i}$ is defined as $q_{(sa)_i} = \langle n, q \rangle$, where $n : string$ is the query name, q is a traditional service query and $q \in \{q_e, q_w\}$, where q_e is an exact traditional query, and q_w is a weighted traditional query.*

The semi-automatic query is formed to meet two main goals. The first goal is to indicate the *ConfiguredService* selected by the service requester from the set of candidate *ConfiguredService* for the previous query. The second goal is is to define the new required functionality. The semi-automatic query structure consists of three parts. The first part contains the unique name defined in the initialization query. The second part is a traditional style query defining the new required functionality and associated nonfunctional, trustworthiness, legal and context information. The third part contains the selected *ConfiguredService* for the previous query.

Definition 19 *The semi-automatic query q_{sa} is defined as $q_{sa} = \langle n, q, cs \rangle$, where $n : string$ is the query name, q is a traditional service query , $q \in \{q_e, q_w\}$ where q_e is an exact traditional query and q_w is a weighted traditional query, and cs is the selected* ConfiguredService *for the previous query response.*

The ending query has to meet two main goals. The first goal is to indicate the *ConfiguredService* selected by the service requester from the set of candidate *ConfiguredService* for the previous query. The second is to inform the Planning Unit that this is the last semi-automatic query which will tell the Planning Unit to compose all previous selections, send the composition result to service requester and end session. The end query is just a regular semi-automatic query with the field q empty.

Example 16 *This example illustrates a semi-automatic composition for a travel planning example. In this example, the service requester is requiring to book a flight and rent a car. To achieve this, the service requester decides to make a semi-automatic composition. He will first select a flight and according to this selection he will rent the car. The composition queries are defines as:*



Name	**TravelPlanning**
Functionality	**Precondition: HasPassport == true** **Postcondition: HasReservation == true** **Domain: AirlineDomain** **Functionality: BookingFunctionality** **Outputs: time**
Nonfunctional	**Price: < 500$ per hour**
Input Parameters	**TravelData:date** **DepartureCity:string** **DestinationCity:string**
Output Parameters	**DepartureAirport:string** **DistinationAirport:string** **Price:double**
Legal Issues	**Deposit <= 50**
Context	**Context Info: [Membership : CAA]**

Figure 36: Semi-automatic Query Initialization

- *The semi-automatic initialization query, shown informally in Figure 36, and is formally defined as:* $q_{sai} = \langle n, q \rangle$ *where* $n =$ "$TravelPlanning$" *and* $q = q_e$ *where* $q_e = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda} \rangle$ *and:*

  - $\hat{f} = \langle \hat{pr}, \hat{po}, \hat{D}, \hat{SF} \rangle$, *where* $\hat{pr} = (HasPassport == true)$, $\hat{po} = (HasRe servation == true)$, $\hat{D} = (AirlineDomain)$, *and* $\hat{SF} = (BookingFuncti onality)$.
  - $\hat{\kappa} = \langle \hat{p} \rangle$, *where* $\hat{p} = \langle \hat{a}, \hat{cu}, \hat{un} \rangle$, $\hat{a} = (500)$, $\hat{cu} = (dollar)$ *and* $\hat{un} = (hour)$.
  - $\hat{l} = \{(deposit = 50)\}$.
  - $\hat{c} = \{(membership == caa)\}$.
  - $\hat{\Lambda_{input}} = \{(TravelDate, date), (DepartureCity, string), (DestinationCity, string)\}$.
  - $\hat{\Lambda_{output}} = \{(DepartureAirport, string), (DestinationAirport, string), (Tr avelTime, time), (Price, double)\}$.

- *The semi-automatic query is formally defined as* $q_{sa} = \langle n, q, cs \rangle$, *where* $n =$ “$Travel\ Planning''$, $cs = Book\_Flight$, *and* $q = q_{e1}$ *where* $q_{e1} = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda} \rangle$ *and:*
  - $\hat{f} = \langle \hat{pr}, \hat{po}, \hat{D}, \hat{SF} \rangle$, *where* $\hat{pr} = (CarBroken == true)$, $\hat{po} = (HasAp pointment == true)$, $\hat{D} = (CarDomain)$, *and* $\hat{SF} = (RentingFunctionality)$.
  - $\hat{\kappa} = \langle \hat{p} \rangle$, *where* $\hat{p} = \langle \hat{a}, \hat{cu}, \hat{un} \rangle$, $\hat{a} = (50)$, $\hat{cu} = (dollar)$ *and* $\hat{un} = (hour)$.
  - $\hat{l} = \{(deposit = 500)\}$.
  - $\hat{c} = \{(membership == caa)\}$.
  - $\hat{\Lambda_{input}} = \{(Duration, time)\}$.
  - $\hat{\Lambda_{output}} = \{(CarType, string), (Price, double)\}$.

- *The ending query is formally defined as* $q_{sa} = \langle n, q, cs \rangle$, *where* $n = "TravelPlanning”$, $q = \phi$, *and* $cs = Rent\_Car$.

8.2.2 Planning Unit Algorithms

In semi-automatic composition, the Planning Unit runs two main processes. The first is concerned with matching and ranking the service requester requirements with candidate

ConfiguredServices. The second is concerned with composing the selected *ConfiguredServices*.

The matching process is performed according to Algorithm 1 where Q is the service query, C is the composition so far, CS is the set of candidate *ConfiguredServices* that provide the functionality required by the query and MATCH algorithm is defined in Section 7.3.3.

Algorithm 1 Semi-automatic Matching

```
INPUT: ServiceQuery ”Q”, CompositionSoFar ”C”, Set of ConfiguredServices ”CS”.
OUTPUT: ServiceType ”ST”.
if C is empty then
    ST=MATCH Algorithm (Q, CS);
    Return ST;
end if
if C is not empty then
    ServiceQuery temp = Q;
    temp preconditions = Q preconditions + C preconditions + C postconditions;
    temp inputParamters = Q inputParamters + C inputsParamters + C outputsParamters;
    ST = MATCH Algorithm (temp, CS);
    Return ST;
end if
```

The result of the semi-automatic matching algorithm is then passed to the ranking algorithm defined in Section 7.3.3. The set of ranked *ConfiguredServices* is then passed to the service requester.

The Planning Unit composes the selected *ConfiguredServices* according to the Semi-automatic composition Algorithm 2, where $A \gg B$ is defined using the semantic discussed in Section 5.2.

Example 17 *The semi-automatic composition algorithm is applied in the previous example. Initially, the composition so far is empty. Next, the composition so far will include* $Book\_Flight$*. Finally, composition is done, and the composition so far will include* $Book\_Flight \gg Rent\_Car$.

Algorithm 2 Semi-automatic Composition

```
INPUT: CompositionSoFar ”C”, ConfiguredService ”S”.
Output: CompositionSoFar ”C’”
if C is empty then
    C’=S;
    Return C’;
end if
if C is not empty then
    C’ = C $\gg$ S;
    Return C’;
end if
```

8.3 Automatic Composition

In automatic service composition, the composition logic is created by the Planning Unit without any input from the service requester. The composition query is just a traditional-style query. In this query, the requester specifies his requirements. These requirements cannot be satisfied by one *ConfiguredService* and hence a composition is necessary. In *FrSeC*, automatic service composition problems are solved as Artificial Intelligence (AI) planning problems. The rest of this section introduces AI planning, transforming service composition into a planning problem and ranking the results of the composition process.

8.3.1 AI Planning

Planning [Lug08] is a branch of AI that has a long history. The main task of planner is to find a sequence of actions that allows the problem solver to achieve some specific task.

Definition 20 *A planning problem can be defined as a tuple $\langle I, A, G\rangle$, where I is the initial state, A is a set of available actions (operations) and G is the set of goals.*

The two main approaches in solving planning problems are situation-space search and planning-space search. Each type depends on the kind of search space that is explored.

- In situation-space [Lug08], the search space is the space of all possible states or situations of the world. The initial state is defined as one node and the goal node is a state where all goals in the goal state are satisfied. The solution of the planning problem will be the sequence of actions in the path from the start node to a goal node.

The two main approaches to situation-space planning are progression planning and regression planning.

- In progression planning, forward-chaining is performed from initial state to goal state. The result will look like a state-space search and any kind of search algorithm such breadth first search or depth first search can be used. The main disadvantage of such approach is the huge number of spaces to explore. Hence, this approach is inefficient. Progression planning can be summarized in the following steps:

  1. Start from initial state.
  2. Find all operations whose preconditions are true in the initial state.
  3. Compute the effects of operators to generate successor states.
  4. Repeat steps 2 and 3 until a new state satisfies the goal conditions.

- In regression planning, backward chaining is performed from the goal state to initial state. This approach is goal directed and usually more efficient than progression planning. The main reason for this is that many operations are available at each state, but only a small number are applicable for achieving a given goal. The main disadvantage of such approach is that it cannot always find a plan even if one exists. Regression planning can be summarized in the following steps:

  1. Start with goal node corresponding to the goal to be achieved.
  2. Choose an operation that will achieve one of the goals.
  3. Replace that goal with the operation preconditions.
  4. Repeat steps 2 and 3 until the initial state is reached.

- In plan-space [Lug08], the search space is the space of all possible plans. A node corresponds to a partial plan. Initially one node in the space will be specified as an “initial plan”. A goal node is a node that contains a plan that satisfies all of the goals in the goal state. This node contains the sequence of actions that determines the solution plan.

8.3.2 Automatic Composition using AI Planning

Service composition involves ordering a set of services in the correct order to satisfy a given goal. This can be viewed as a planning problem P described by the tuple $\langle I, A, G\rangle$ where I is the initial state, A is the set of actions and G is the goal. The planning problem can be mapped to service composition as following:

- Initial state is replaced by the preconditions and input parameters provided by the service requester.
- The set of actions are replaced by the available *ConfiguredServices* in the related domains.
- The goal is replaced by the required preconditions and output parameters required by the service requester.

To do the planning we suggest the use of regression planning discussed earlier. We will use the STRIPS [Lug08] planner as an example.

STRIPS represents states as sets of atomic facts. The set A contains all the actions that can be used to modify states. Each actions A_i has three lists of facts containing the preconditions of A_i defined as $prec(A_i)$, the facts that are added (postconditions) by A_i defined as $add(A_i)$, and the facts that are deleted from the world state after the application of A_i defined as $del(A_i)$. The following rules fold for the states in STRIPS:

- An action A_i is applicable to state S if $prec(A_i) \subseteq S$.
- If A_i is applied to S, the resulted state $\grave{S}$ is defined as $\grave{S} = S = del(A_i) \cup add(A_i)$.
- The solution to the planning problem is a sequence of actions $P = A_1, A_2, ..., A_n$, which if applied to I results to a state $\grave{S}$ such that $G \subseteq \grave{S}$.

The Planning Domain Definition Language (PDDL) [Kov] is an attempt to standardize planning domain and problem description languages. It was first developed by Drew McDermott in 1998 and later has been enhanced, extended and became a standard for modeling planning domains and problems. The latest version of this language is PDDL3.1. PDDL can be used to represent all the elements of STRIPS. A planning problem specified in PDDL is divided into two files. The first file specifies the domain file for predicates and actions. The second file specifies the problem including objects, initial state and goal specification.

We will use PDDL to formalize our planning problem. Translating our service composition problem into a planning problem defined in PDDL includes two steps.

STEP 1: Defining the planning domain file

This step translates available *ConfiguredServices* defined using SQL (Chapter 9) into a planning action. Each *ConfiguredService* CS_i is mapped into an action A_i according to the following rules:

- The name of action $name(A_i)$ is mapped to the *ConfiguredService* name $name(CS_i)$.

- The preconditions of the actions are the union of the *ConfiguredService* input parameters, preconditions and context rules.

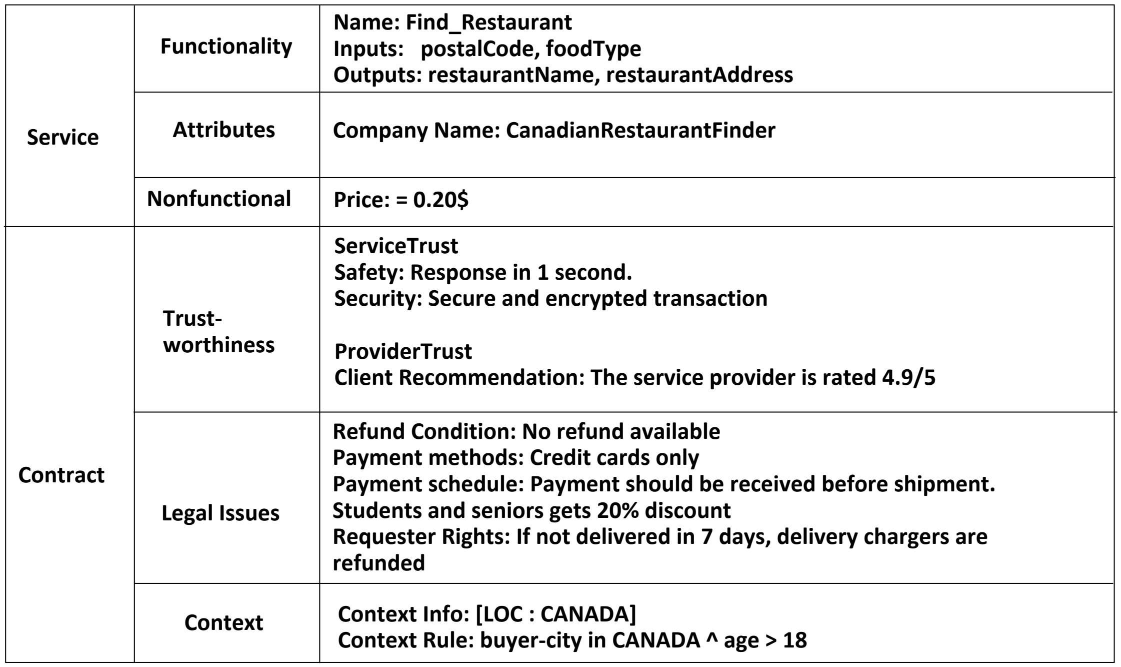
  $prec(A_i) = CS_i.inputParameters \cup CS_i.preconditions \cup CS_i.contextRules.$

  It is essential to note that input parameters consist of variables ($Invar$) and value ($value$). These parameters can be written as logical statements in the form $Invar == value$. This will ensure that $prec$ is a logical statement.

- The *add* effects of the action are the union of the *ConfiguredService* output parameters and postconditions.

  $add(A_i) = CS_i.outputParameters \cup CS_i.postconditions.$

  It is essential to note that output parameters consist of variable ($Outvar$) and values ($value$). These parameters can be written as logical statements in the form



Service	Functionality	Name: Find_Restaurant Inputs: postalCode, foodType Outputs: restaurantName, restaurantAddress
	Attributes	Company Name: CanadianRestaurantFinder
	Nonfunctional	Price: = 0.20$
Contract	Trust-worthiness	ServiceTrust Safety: Response in 1 second. Security: Secure and encrypted transaction ProviderTrust Client Recommendation: The service provider is rated 4.9/5
	Legal Issues	Refund Condition: No refund available Payment methods: Credit cards only Payment schedule: Payment should be received before shipment. Students and seniors gets 20% discount Requester Rights: If not delivered in 7 days, delivery chargers are refunded
	Context	Context Info: [LOC : CANADA] Context Rule: buyer-city in CANADA ^ age > 18

Figure 37: Find_Restaurant ConfiguredService

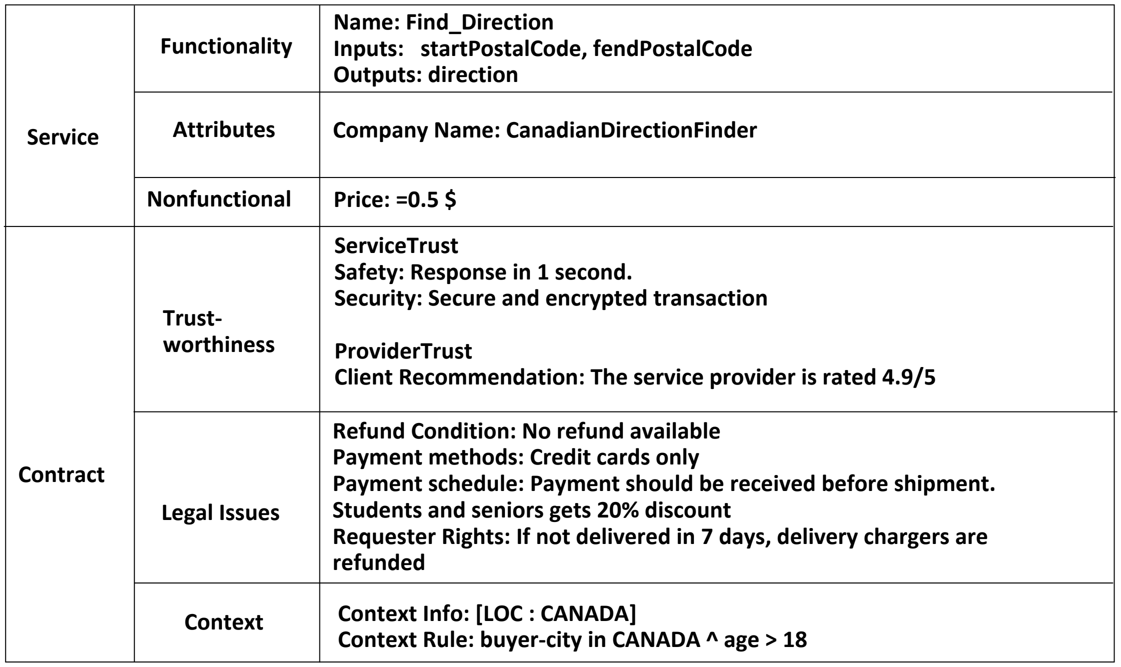
$Outvar == value$. This will ensure the add is a logical statement.

- The delete list will remain empty.

Example 18 *Figures 37, 38 and 39 illustrate three* ConfiguredServices. *The first* ConfiguredService *finds the nearest restaurant to a specific postal code using food type. An example would be finding an Italian restaurant nearest to a location, given its postal code N1N1N1. The second* ConfiguredService *will find the direction between two locations, given their postal addresses. The third* ConfiguredService *will estimate the time required to travel a predefined direction. These* ConfiguredServices *can be translated into the PDDL domain file presented in Figure 40.*

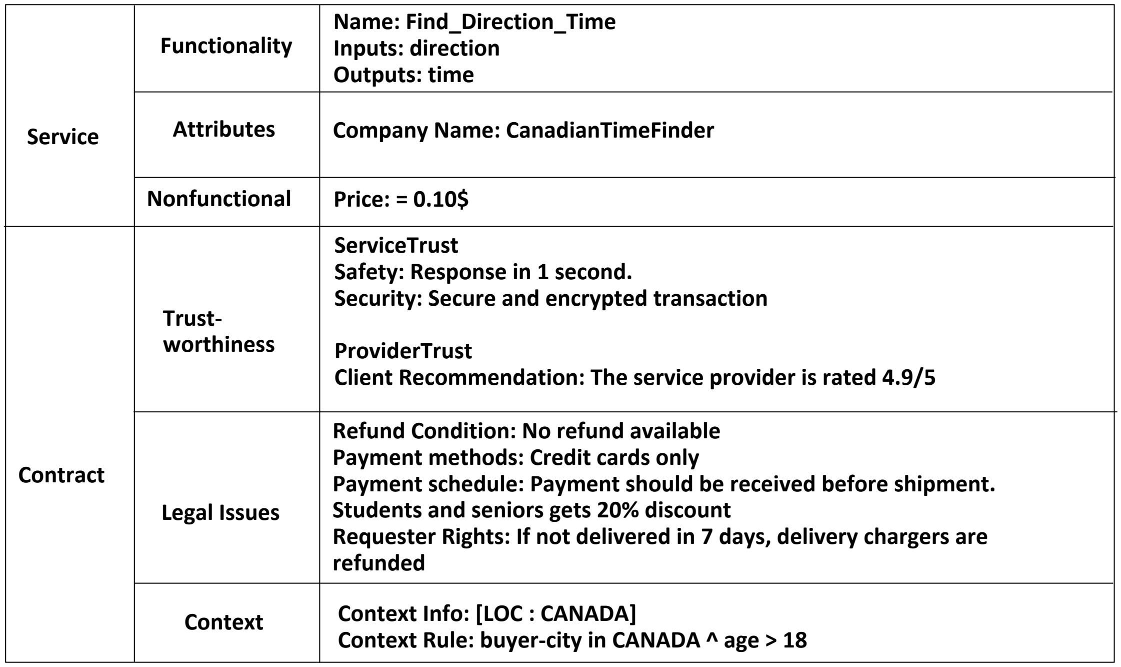
STEP 2: Defining the problem file

This step translates a query Q defined in SQL into the problem file according to the following rules:



Service	Functionality	Name: Find_Direction Inputs: startPostalCode, fendPostalCode Outputs: direction
	Attributes	Company Name: CanadianDirectionFinder
	Nonfunctional	Price: =0.5 $
Contract	Trust-worthiness	ServiceTrust Safety: Response in 1 second. Security: Secure and encrypted transaction ProviderTrust Client Recommendation: The service provider is rated 4.9/5
	Legal Issues	Refund Condition: No refund available Payment methods: Credit cards only Payment schedule: Payment should be received before shipment. Students and seniors gets 20% discount Requester Rights: If not delivered in 7 days, delivery chargers are refunded
	Context	Context Info: [LOC : CANADA] Context Rule: buyer-city in CANADA ^ age > 18

Figure 38: Find_Direction ConfiguredService



Service	Functionality	Name: Find_Direction_Time Inputs: direction Outputs: time
	Attributes	Company Name: CanadianTimeFinder
	Nonfunctional	Price: = 0.10$
Contract	Trust-worthiness	ServiceTrust Safety: Response in 1 second. Security: Secure and encrypted transaction ProviderTrust Client Recommendation: The service provider is rated 4.9/5
	Legal Issues	Refund Condition: No refund available Payment methods: Credit cards only Payment schedule: Payment should be received before shipment. Students and seniors gets 20% discount Requester Rights: If not delivered in 7 days, delivery chargers are refunded
	Context	Context Info: [LOC : CANADA] Context Rule: buyer-city in CANADA ^ age > 18

Figure 39: Find_DirectionTime ConfiguredService

```
(define (domain example1)

  (: action FindRestaurant
   : parameters (?postalCode ?foodType ?city ?age
                  ?restaurantName ?restaurantPostalCode)
   : precondition ( and (not (= null ?postalCode))
                        (not (= null ?foodType))
                        (in CANADA ?city)
                        (< 18 age))
   :effect (and (not (= null ?restaurantName))
                (not (= null ?restaurantPostalCode))))

  (: action FindDirectionRestaurant
   : parameters (?startPostalCode ?endPostalCode ?city
                 ?age ?direction)
   : precondition ( and (not (= null ?startpostalCode))
                        (in CANADA ?city)
                        (< 18 age))
   :effect (and (not (= null ?direction))))

  (: action FindDirectionTime
   : parameters (?direction ?city ?age ?time)
   : precondition ( and (not (= null ?direction))
                        (in CANADA ?city)
                        (< 18 age))
   :effect (and (not (= null ?time))))
)
```

Figure 40: PDDL for Example 18 ConfiguredServices

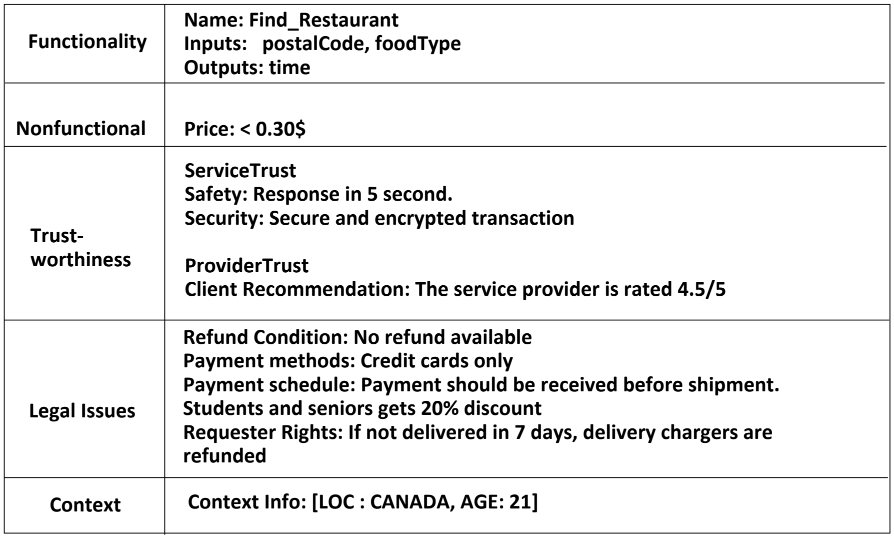
- The initial state is defined as the union of input parameters, the preconditions and context information defined in the query.

  $I = Q.inputParamters \cup Q.preconditions \cup Q.contextInfo.$

- The goal state will be the union of the required postconditions and output parameters as defined in the query.

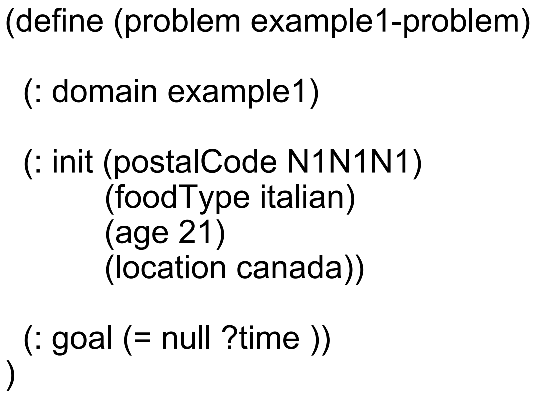
  $G = Q.postconditions \cup Q.outputParameters.$

Example 19 *Figure 41 shows a service query. This query is translated to the PDDL problem file presented in Figure 42 according to the rules presented above.*



Functionality	**Name: Find_Restaurant** **Inputs: postalCode, foodType** **Outputs: time**
Nonfunctional	**Price: < 0.30$**
Trust-worthiness	**ServiceTrust** **Safety: Response in 5 second.** **Security: Secure and encrypted transaction** **ProviderTrust** **Client Recommendation: The service provider is rated 4.5/5**
Legal Issues	**Refund Condition: No refund available** **Payment methods: Credit cards only** **Payment schedule: Payment should be received before shipment.** **Students and seniors gets 20% discount** **Requester Rights: If not delivered in 7 days, delivery chargers are refunded**
Context	**Context Info: [LOC : CANADA, AGE: 21]**

Figure 41: Example 19 Query



```
(define (problem example1-problem)

  (: domain example1)

  (: init (postalCode N1N1N1)
          (foodType italian)
          (age 21)
          (location canada))

  (: goal (= null ?time ))
)
```

Figure 42: Example 19 PDDL problem file

8.3.3 Mapping Planning Result to Service Composition

The result of the planning problem is a set of ordered actions. These set of actions are translated into a service composition. The service composition will be a sequential composition of the ordered actions. Where each action is replaced with the corresponding *ConfiguredService*.

Example 20 *The plan resulting from the previous example will consist of the three actions* FindResturant, FindDirection, FindDirectionTime. *This can be mapped to the composition* $Find\_Restaurant \gg Find\_Direction \gg Find\_Direction\_Time$.

8.3.4 Composition Matching and Ranking

In general, the Planning Unit will generate multiple candidate service composition plans. Using the static service composition constructs and their semantics, discussed in Section 5.2, the composition is mapped in a one-to-one manner into a single *ConfiguredService*. Hence, the candidate plans are now represented by a set of candidate *ConfiguerdServices* where each *ConfiguredService* is resulted from a single plan.

The resulting *ConfiguredServices* (resulted from the plans) provide the required functionalities but they might not satisfy the required nonfunctional, trustworthiness and legal requirements. Hence, a matching should be performed between the service requesters requirements and the resulted *ConfiguredServices*. The matching algorithm discussed in Chapter 7 will be used.

The service requester query might have been an exact or partial matching query. In case of a partial matching query, the ranking algorithm in Section 7.3.3 will be called. This step will filter out all compositions that do not satisfy the service requester nonfunctional, trustworthiness and legal requirements.

The result of the matching and ranking process will be a set of ordered *ConfiguredService* where each *ConfiguredService* represent a single composition plan. The ranked plans are then passed to the service requester.

8.4 Summary

This chapter has introduced three novel dynamic service composition approaches that are supported by *FrSeC*. The first approach is template-based, where the service requester defines a composition by defining the requirements and the execution logic of the matched *ConfiguredServices*. The second approach is semi-automatic, where the service requester has continues interaction with the Planning Unit to select services that will be part of the composition created by the Planning Unit. The third approach is automatic, where the service requester specifies a requirement that cannot be matched by a single functionality and the Planning Unit is responsible for creating a composition plan to meet the service requester requirements. The service requester has the choice to select any dynamic composition approach. If the template-based composition used sequential composition constructors only, all three dynamic composition approaches will result in the same service composition. In term of complexity, the automatic composition approach is the most complex. This is due to the fact that a huge number of services and options have to be analyzed by the Planning Unit. Hence, the main limitations of the automatic composition approach is its scalability and performance.

Chapter 9

Development Stages and Supporting Languages

This thesis has introduced a new service model, a new composition theory, and a new provision framework *(FrSeC)* that enables the provision and composition of services defined using the new service model. This chapter shows how service-oriented applications can be developed according to *FrSeC* principles.

The development of service-oriented applications includes multiple stages. Section 9.1 identifies these stages and their relation to *FrSeC*.

The approach proposed for the development of service-oriented applications is formally based. The use of formal methods is complex and error prone. Therefore, there is a need for a set of languages that enable the participants at the different development stages to fulfill their responsibility without worrying about complex formal languages. Hence, this chapter introduces a set of languages necessary to support the development of service-oriented applications. The languages are defining to support each step of the development. These languages were designed to ensure semantic simplicity, modularity, composability and extendibility. Semantic simplicity will increase the languages usability and comprehension. Modularity will lower the coupling and increase the re-usability of languages definitions. Composability will enable complex units to be built easily by composing simpler units. Extendibility will ensure that the languages are able to develop and adapt in the future.

This chapter also introduces five XML-based languages which are used for the transmission of data between the different components of *FrSeC*. These XML-based languages are not to be used by users of *FrSeC* and hence the users don't have to worry about the complexity of XML. The details of the introduced languages are presented in Section 9.2. In Section 9.3 we illustrate the use of the proposed languages on the auto roadside emergency service case study.

9.1 Service-oriented Application Development Stages

A service-oriented application is an application that uses services as the means for providing functionality. It is usually a composition of interacting services to provide a new complex functionality that cannot be provided by a single atomic service. The development of service-oriented applications using *FrSeS* involves four main stages, namely service definition, service implementation, service processing, and service provision. The provision stage also includes service composition. Below we give a brief review of what we have done in these stages and follow it up with a discussion of the languages necessary to carry out the activities of these stages.

9.1.1 Service Definition

Services are defined by service providers in a contract first approach. That is, the contract is defined before the implementation of service [Erl07]. At this stage, the service provider determines all the possible contracts that this service should satisfy. The contract will include the guarantees the service provider can make when providing the service functionality. Guarantees are associated with legal and trustworthiness issues.

In Chapter 4, we have presented *ConfiguredService* which is a structure that associates a service with a contract. The *ConfiguredService* service part will include information about service function, nonfunctional properties and attributes. The contract part will contain information regarding the service provision legal rules, the trustworthiness guarantees and the contextual rules constraining the service provision. The *ConfiguredService* structure can be used by service providers to define their services at the service definition stage.

9.1.2 Service Implementation

A single functionality may be associated with multiple contracts. Hence, multiple *ConfiguredServices* can provide the same functionality but with different contracts. Such *ConfiguredServices* might have one implementation which we call *ImplementedService*.

After defining the *ConfiguredServices*, the service provider develop the *ImplementedService* that implements these *ConfiguredServices*. Implementing services can be performed following different software engineering methodologies. We propose the use of the formal software engineering methodology for the development of trustworthy component based systems (TADL) [MA11].

9.1.3 Service Provision and Processing

In *FrSeC*, service provision includes four steps: (1) service publication, (2) service discovery, (3) service execution and (4) service delivery. The two main interacting elements in service provision are service providers and service requesters. Below is a brief discussion of the service provision stages.

Service Publication

During service publication a service provider prepares a service for publication. The published information should enable the discovery and selection of the service. The published information will include the functionality the service can provider, the nonfunctional and trustworthiness properties guaranteed by the service, the legal rules and exceptions guaranteed and required by the service provider, and the contextual information of the service provider. In addition to the previous information, the service description should also include a set of context rules. The context rules define the context in which the service provider can guarantee the information define in its service description. This context will be the context of the service requester, consumer and service provision. Hence, the *ConfiguredService* definition is sufficient for service publication.

In *FrSeC*, the publication of *ConfiguredServices* is done through the Service Registry. It provides a medium for the publication of *ConfiguredServices* and any related semantic information. The Service Registry will only accept the publication of *ConfiguredServices* that has been analyzed and approved by the Trusted Authority. We call this analysis *the before publication analysis* and it has been discussed in Chapter 4.

During service publication the service consumer plays no role. But the service provider might use consumers' contextual information to constrain the service. This is done by defining context rules.

Service Discovery

In the discovery process, service requesters are looking for services that can provide their requirements. But these requirements are not always clear and precise. Hence, in *FrSeC* we have the two types of discovery process, called traditional and buffet (Section 7.3.2.

In the traditional style, the service requester browses the Service Registry for available functionalities, and semantic information. The information found in the Registry will then be used by the service requester in defining their queries. Because different requesters has different rules, *FrSeC* introduced two types of queries exact and weighted. The queries are sent to the Planning Unit which will communicate with the Service Registry to determine the *ConfiguredServices* that matches their requirements. But in many cases there are multiple matches and the service requester might not be able to decide which one is more appropriate. Hence, a ranking algorithm is used by the Planning Unit of the *FrSeC* (Section 7.3.3).

In the buffet style, the service requester browses the Service Registry for available *ConfiguredServices*. And when defining the query a specific *ConfiguredService* is requested rather than general functionality.

Even after a *ConfiguredService* has been selected by the service requester some negotiation with the service requester might be necessary. Hence, in *FrSeC* we have introduced the Plan Negotiation Unit. This is responsible for mediating the negotiation between the service requesters and service providers.

Service Execution

In *FrSeC*, service execution is managed by the Execution Unit. At the beginning of service execution, the Execution Unit receives from the service requester (1) selected *ConfiguredService* and (2) consumer data and context. The Execution Unit then generates a new service agreement that includes (1) the original *ConfiguredService*, and (2) the service requester data and context. The new agreement we call an *ExecutableService*. In the *ExecutableService* only the legal rules that apply to the service requester are left in the contract. In other words, the *ExecutableService* contract is tailored to the service requester.

The new *ExecutableService* is then sent to both the service requester and the service provider. Both parties have to accept it. After acceptance the Execution Unit will execute the *ExecutableService*. The execution is performed by sending service requests and receiving service responses to and from service providers.

Service Delivery

After the service has been delivered service providers and requesters should verify the execution of the service. This verification will ensure that neither party has violated his obligations.

The service requester will verify that the service provider (1) has provided the requirement that satisfies the postconditions, (2) has fulfilled the obligations stated in the contract, and (3) did not violate the agreed upon legal rules. In *FrSeC*, the requester verification can be performed by service requester himself or it can be delegated to the Trusted Authority. If the verification shows a violation from the service provider, the service requester will consult the legal rules to find the penalties for the violation. It is the responsibility of the service requester to prove that the service provider did violate the agreed upon contract. The need for such proof highlights the need for the Trusted Authority.

The service provider also verifies that the service requester fulfilled his obligations.

This verification can also be delegated to the Trusted Authority. If the service requester fulfilled his obligations, the service provider will close the contract and end the process. Otherwise, it may define additional requirements.

9.1.4 Service Composition

A service-oriented application can be thought of as a composite service. Service composition may be attempted either at design-time or at execution-time. The former, called *static* composition and the later called *dynamic* service composition. *FrSeC* supports static and dynamic service composition in the development of applications.

Static service composition is provider-driven. The result of the composition is a *ConfiguredService* that is published using the Service Registry. Dynamic service composition is requester-driven. Dynamic service composition can be of the three types template-based, semi-automatic and automatic (Chapter 8). All three types can be used to create service-oriented application using *FrSeC*.

9.2 FrSeC Supporting Languages

In developing a service-oriented application, services are to be made first class citizens. This implies that services, regarded as first class objects, are created, announced, discovered, executed, and delivered. To meet these activities, languages that are easy to use must exist. Figure 43 shows the languages in a certain hierarchy, and below we discuss their design merits.

9.2.1 Service Processing Languages (SPL)

After some investigation we found that a unified syntax can be used for languages that are required for the different stages of a service application development. So, we designed the language Service Processing Languages (SPL). Because of its generic structure it is indeed a family of languages. The full syntax of SPL family is presented in Appendix A.

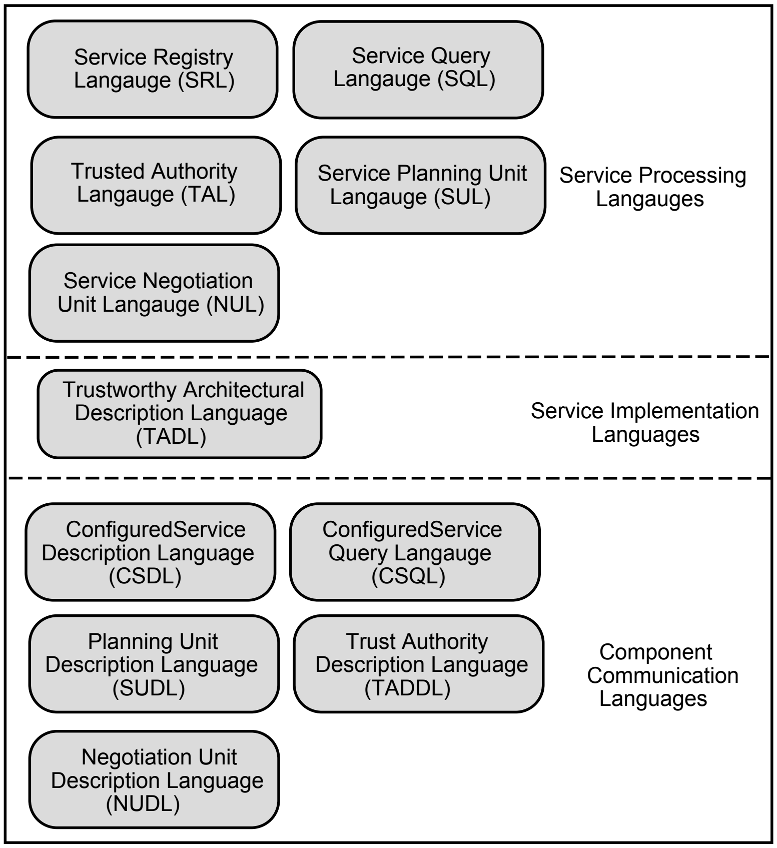

Service Registry Langauge (SRL)
Service Query Langauge (SQL)
Trusted Authority Langauge (TAL)
Service Planning Unit Langauge (SUL)
Service Processing Langauges
Service Negotiation Unit Langauge (NUL)
Trustworthy Architectural Description Language (TADL)
Service Implementation Languages
ConfiguredService Description Language (CSDL)
ConfiguredService Query Langauge (CSQL)
Planning Unit Description Language (SUDL)
Trust Authority Description Language (TADDL)
Component Communication Languages
Negotiation Unit Description Language (NUDL)

Figure 43: Language Support

In SPL the syntax is such that every element is described *separately*. The rationale behind this is two-fold. First, it increases *reuse* of existing elements for different specifications. For example, a context information element may be reused in different *ConfiguredServices*. Therefore, having the context information specification described separately from *ConfiguredService* specification will enable reuse. Second, *reconfiguration* of the elements affect only locally. For example, more information can be added to an existing context specification by changing only the context information. The main features of SPL are summarized below.

- SPL syntax is close to our conceptual view of the architectural elements and hence easy to understand.

- Formal semantics can be given to the information specified in it. The semantic basis is provided by set theory and logic, as described in Chapter 7. Abstract data types are modeled by sets and constraints are expressed in first order logic.

- The language is extendable, in the sense that when more architectural elements are introduced, more syntactic units can be added to SPL syntax.

- The language allows heterogeneous collection of elements to be either grouped or described as independent modular units. Basic and abstract data types can be included to enrich the elements.

- A syntactic unit (modular unit) can be included in another unit. Consequently larger specification units are built up incrementally.

The syntax for describing an element contains element type, element name, and a specification of the contents of the element. Figure 44 shows the general elements that are used by SPL. Those general elements are attributes and parameters. We use attributes with every elements of SPL. Attributes can be used to specify any semantic information to be associated with any SPL element. Parameters are used to define data parameters. A data parameter is defined in terms of it name and data type. Note that $(Attribute < name >)^\star$ means that 0 or more attributes can be defined as part of the element.

```
Attribute < name > {
        < DataType >< name >;
        Default < value >;
}

ParameterType < name > {
        (Attribute < name >)*;
        < DataType >< name >;
}
```

Figure 44: SPL General Elements Syntax

SPL family includes the following languages that we need for different stages in developing a service-oriented application:

- Service Registry Language (SRL): It is used to describe the service registry hierarchically. The syntax of the language SRL, presented in Appendix A.1, is a concrete version of the abstract formal notation discussed in Chapter 7. Consequently, the semantic basis of SRL is formal. Service providers describe *ConfiguredServices* in the language CSL in order to publish them in the Service Registry of the *FrSeC*. There is a close affinity between the structure of the Service Registry and the structure of a *ConfiguredService*. This affinity implies that the SRL syntax must be a superset of the CSL syntax. The concrete syntax of the language CSL is based on a meta-model derived from the formal definition of *ConfiguredService*. The meta-model and the syntax are illustrated in Appendix A.1.

- Service Query Language (SQL): A family of query languages is needed to prepare traditional queries, buffet queries, and composition queries. The query language that is necessary for a transaction is prepared by the PU and given to service requesters to describe service requests. The syntax for each query language has an underlying meta-model. The meta-models and the complete syntax of SQL are given in Appendix A.2.

- Trusted Authority Language (TAL): This language is defined by the TA. It is used to describe authentication certificate requests and analysis requests sent to the Trusted Authority. It is used by both service providers and service requesters. The full syntax of TAL appears in Appendix A.3.

- Service Negotiation Unit Language (NUL): This language is defined by the Negotiation Unit. It is used by service requesters to request service negotiations and used by the Negotiation Unit to convey the negotiation result back to requesters. The full syntax of NUL appears in Appendix A.5.

- Service Planning Unit Language (SUL): This language, defined by the PU, will be used to describe service lookups, lookup results and service plans. It is used by the Planning Unit. The complete syntax of SUL is given in Appendix A.4.

9.2.2 Service Implementation Languages (TADL)

In principle a service provider can use any implementation strategy to implement the CSL descriptions. We propose the use of the formal component-based software engineering methodology for the development of trustworthy systems [MA11]. In this work a trustworthy architecture description language, called TADL, has been formally defined. We use TADL as the service implementation language because of the following merits.

- it is quite generic and expressive,
- it is formal and its descriptions can be formally verified,
- trustworthiness properties can be described in it, and
- it comes with tools, such as the visual modeling tool (VMT) [Moh09] for design time development and the transformer [Ibr08] for linking to UPPAAL.

So, it is sufficient to explain now the mapping of a *ConfiguredService* to a TADL template. The result of mapping will be called an *ImplementedService*.

An *ImplementedService* is one TADL component (primitive or composite). It can encapsulate one or more *ConfiguredServiecs*. A *ConfiguredService* is mapped to a TADL component service according to the following rules. In Figure 45 the left side is *ConfiguredService* and the right side is the 'ComponentType' of TADL which represents the *ImplementedService*.

- **Mapping Service Part**: The service part of a *ConfiguredService* is mapped to the service part *Service* of the TADL template.
  - **Function**: Each function is mapped to TADL component service as follows:
    - The preconditions are mapped to data constraints.
    - The postconditions are mapped to TADL component service update statements.
    - The signature id is mapped to TADL component service stimulus.

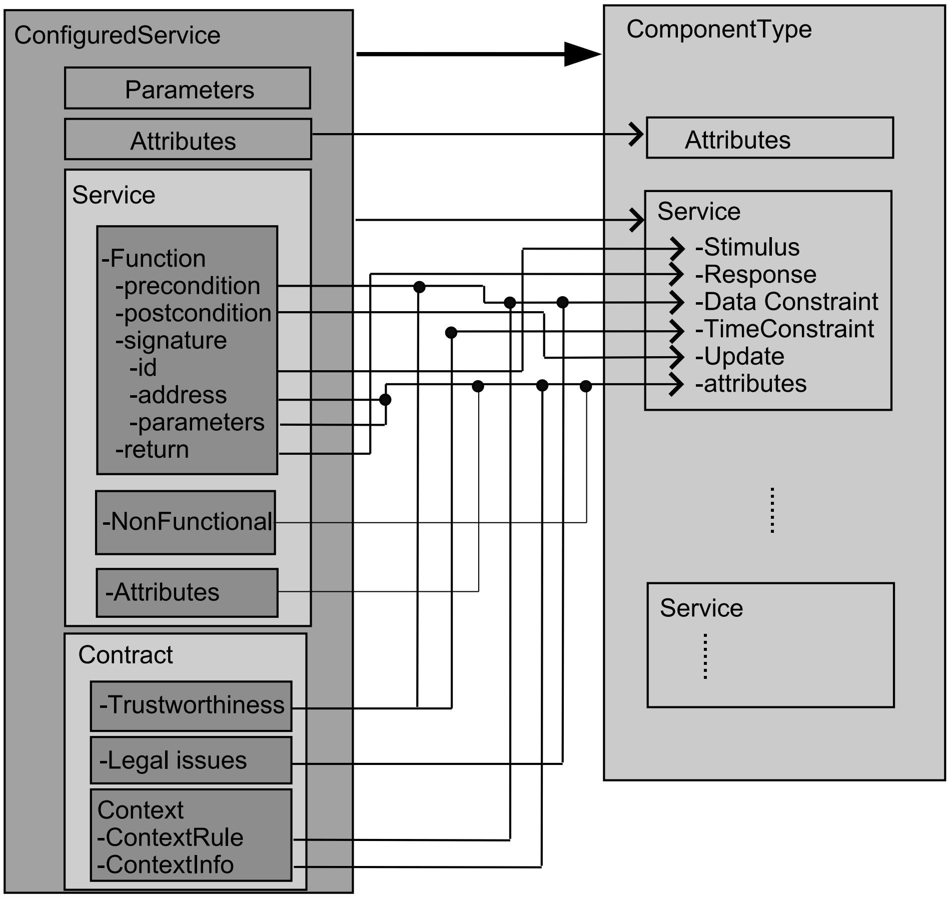


Figure 45: ConfiguredService and ComponentType mapping

  ∗ The signature address is mapped to TADL component service attributes.

  ∗ The signature parameter is mapped to TADL component service attributes.

  – **Nonfunctional**: The nonfunctional properties are mapped to attributes in the TADL component service.

  – **Attributes**: The attributes are mapped to TADL component attributes.

- **Contract**: The elements of the contract are mapped to data constraints, time constraints, and attributes of the ‘ComponentTemplate’.

  – **Trustworthiness**: The trustworthiness properties are mapped as data and time constrains in the TADL component service.

- **Legal Issues**: The legal rules are mapped as data and time constrains in the TADL component service.
- **Context**: The ContextRules are mapped to data constraints in the TADL component service. The ContextInfo are mapped to the TADL component service attributes.

Thus, every *ConfiguredService* published in the Service Registry has a description in CSL, and the CSL description is mapped faithfully to a TADL component which implements it, we claim that traceability between implementation and its corresponding specification becomes easier. We can develop a tool that will check whether or not (1) every published service is implemented as a TADL component, and (2) every TADL component implementation corresponds to some published service.

9.2.3 Component Communication Languages (CCL)

Service descriptions and service requests are required for both human consumption and for machine computation. The languages CSL, SQL, NUL, TAL and SUL are meant for humans. We need their equivalent machine understandable versions for inter-module communication. When a *ConfiguredService* or query is to be passed as data between two *FrSeC* units (equivalently between two TADL components), we want to have *interoperability*. This is best achieved by the respective XML versions of CSL, SQL, NUL, TAL and SUL.

The XML version of CSL is called *ConfiguredService Description Language (CSDL)*, the XML version of SQL is called *ConfiguredService Query Language* (CSQL), the XML version of NUL is called *Negotiation Unit Description Language (NUDL)*, the XML version of TAL is called *Trusted Authority Description Language (TADDL)*, and the XML version of SUL is called *Planning Unit Description Language (SUDL)*. These languages are faithful XML translations of their respective languages. These translations are automatically driven by a grammar-driven transformation tool. The users do not have to worry about writing complex XML definitions. The users describe their requirements at different stages only in CSL, SQL, NUL, TAL and SUL. The syntax of CSDL is presented in

Appendix B and the syntax of CSQL is presented in Appendix C. The syntax of NUDL, TADDL and PUDL are defined in the same manner.

9.3 A Partial Specification of the Auto Roadside Emergency Case Study

The full case study described in Chapter 7 requires many specifications. We choose three of them for presentation in this section.

This section, illustrates the languages presented in the previous section by presenting a partial specification of the case study described in Chapter 7. The rest of the section is organized as follows. First, the Service Registry part related to service provider Garage1 is specified using SRL. This also includes the specification of *ConfiguredService* garage1.1. Second, the specification of *ConfiguredService* garage1.1 is presented using CSDL. Third, to illustrate the specification of queries, a complete specification of Query1 is presented in SQL and CSQL. Finally, for Query1 the specification of the service lookups, lookups result and generated plan are presented in SUL.

9.3.1 Service Registry Specification

This specification contains only the service registry part used by the service provider Garage1 providing *ConfiguredService* garage1.1, garage1.2 and garage1.3. The specification includes the specification of *ConfiguredServices* garage 1.1. *ConfiguredServies* 1.2 and 1.3 is can be specified in the same manner. The specification is presented in the language SRL.

```
Registry ServiceRegistry{
        RepairShopDomain repairshopdomain;
}
Domain RepairShopDomain{
        ReserveFunctionality reservefunctionality;
        RepairShopSafety repairshopsafety;
```

```
}
DomainNonFunctional RepairShopSafety{
        propertyName safety;
}
Functionality ReserveFunctionality{
        Garage1Node garage1node;
        Garage2Node garage2node;
        Garage3Node garage3node;
        CarBroken carBroken;
        Deposit deposit;
        CarType carType;
        HasAppointment hasAppointment;
        NumOfHours numOfHours;
}
ServiceProviderNode Garage1Node{
        ServiceProviderID garage1;
        Garage1.1 garage1-1;
        Garage1.2 garage1-2;
        Garage1.3 garage1-2;
}
ServiceProviderNode Garage2Node{
        ServiceProviderID garage2;
        Garage2.1 garage2-1;
}
ServiceProviderNode Garage3Node{
        ServiceProviderID garage3;
        Garage3.1 garage3-1;
}
Attribue GarageName{
        DataType String;
        Default Garage1;
}
ParameterType CarBroken{
        DataType Boolean;
}
ParameterType Deposit{
        DataType double;
}
ParameterType CarType{
        DataType string;
}
ParameterType HadAppointment{
        DataType Boolean;
}
ParameterType NumOfHours{
        DataType int;
}
Signature ReserveSignature{
        id RESERVE;
        CarBroken carBroken;
        Deposit depsoit;
        CarType carType;
        address 5141111112; }
Return ReserveReturn{
        id ReserveConfirmation;
        HasAppointment hasAppointment;
        NumOfHours numOfHours;
}
```

```
Precondition ReservePrecondition{
        $CarBroken == TRUE$;
        $HasAppointment == FALSE$;
}
Postcondition ReservePostcondition{
        $HasAppointment == TRUE$;
}
Function Garage1-1Reserve{
        ReserveSignature reservesignature;
        ReserveReturn reservereturn;
        ReservePrecondition reserveprecondition;
        ReservePostcondition reservepostcondition;
}
NonFunctional Garage1-1Reserve-NF{
        Garage1-1Price garage1-1price;
}
Price Garage1-1Price{
        $value$ 60;
        $currency\ CanadianDollar$;
        $unit\ hour$;
        Reserve reserve;
}
Service Garage1-1Service{
        Garage1-1Reserve garage1-1reserve;
        Garage1-1Reserve-NF garage1-1reserve-nf;
}
Trustworthiness Garage1-1Trust{
        ReserveSafety reserveSafety;
}
```

```
Safety ReserveSafety{
        Reserve reserve;
        $maxTime < 7days$;
}
LegalIssue Garage1-1Legal{
        Garage1-1PriceCondition g1-1pc;
        Garage1-1Deposit g1-1d;
        Garage1-1PaymentMethod g1-1pm;
}
PriceCondition Garage1-1PriceCondition{
        Garage1-1Price garage1-1price;
        $CarType == toyota$;
}
DepositRule Garage1-1Deposit{
        $amount$ 300;
        $currency\ CanadianDollar$;
}
PaymentMethod Garage1-1PaymentMethods{
        $paymentMethod\ cash$;
}
ContextRule Garage1-1ContextRule{
        $WhoRequester.membership == CAA$;
        Garage1-1Reserve reserve;
}
ContextInfo Garage1-1ContextInfo{
        Garage1Location garage1location;
}
Location Garage1Location{
        Garage1City garage1city;
        Garage1suburb garage1suburb;
}
Region Garage1-1City{
        type city;
        name montreal;
}
Region Garage1-1Suburb{
        type suburb;
        name downtown;
}
Context Garage1-1Context{
        Garage1-1ContextRule garage1-1contextrule;
        Garage1-1ContextInfo garage1-1contextinfo;
}
Contract Garage1-1Contract{
        Garage1-1Trust trust;
        Garage1-1Legal legalrule;
        Garage1-1Context context;
}
ConfiguredService Garage1-1{
        GarageName garagename;
        Garage1-1Service garage1-1service;
        Garage1-1Contract garage1-1contract;
}
```

9.3.2 CSDL Specification

Below is the CSDL specification of *ConfiguredService* garage1.1 whose SRL specification is presented above.

```
<?xml version="1.0" encoding="UTF−8"?>
<CSDL>
 <Service>
  <Function>
   <Signature>
    <ID>RESERVE</ID>
    <Address>5141111112</Address>
    <Parameter>
     <Name>CarBroken </Name>
     <DataType>bool </DataType>
     <DefaultValue></DefaultValue>
    </Parameter>
    <Parameter>
     <Name>Deposit </Name>
     <DataType>double </DataType>
     <DefaultValue></DefaultValue>
    </Parameter>
    <Parameter>
     <Name>CarType </Name>
     <DataType>string </DataType>
     <DefaultValue></DefaultValue>
    </Parameter>
   </Signature>
   <Return>
    <ID>ReserveConfirmation </ID>
    <Parameter>
     <Name>HasAppointment </Name>
     <DataType>bool </DataType>
     <DefaultValue></DefaultValue>
    </Parameter>
    <Parameter>
     <Name>NumOfHours </Name>
     <DataType>int </DataType>
     <DefaultValue></DefaultValue>
    </Parameter>
   </Return>
   <Precondition>
    <Condition>CarBroken==T &&
    HasAppointment==F</Condition>
   </Precondition>
   <Postcondition>
    <Condition>HasAppointment==T</Condition>
   </Postcondition>
  </Function>
  <NonFunctional>
   <Price>
    <value>60</value>
    <currecny>dollar </currecny>
    <unit>hour </unit>
   </Price>
  </NonFunctional>
 </Service>
 <Contract>
  <Trustworthiness>
   <ConfiguredServiceTrust>
    <Safety>
     <maxTime>7days </maxTime>
    </Safety>
   </ConfiguredServiceTrust>
  </Trustworthiness>
  <Legal>
   <PriceCondition>
    <Price>
     <value>60</value>
     <currecny>dollar </currecny>
     <unit>hour </unit>
    </Price>
    <Condition>CarType==toyota </Condition>
   </PriceCondition>
   <DepositRule>
    <Amount>300</Amount>
    <Currency>dollar </Currency>
    <Rule>String </Rule>
    <Date>2011−07−30</Date>
    <Time>00:00:00 </Time>
   </DepositRule>
   <PaymentRules>
    <PaymentMethod>
      <Method>cash </Method>
    </PaymentMethod>
   </PaymentRules>
  </Legal>
  <Context>
   <ContextInfo>
    <Location>
     <Region>
      <Type>City </Type>
      <Name>Montreal </Name>
```

```
        </Region>
        <Region>
          <Type>Suburb </Type>
          <Name>Downtown </Name>
        </Region>
      </Location>
    </ContextInfo>
    <RequesterContextRules>
      <WhoRequester>
        <Membership>CAA</Membership>
      </WhoRequester>
    </RequesterContextRules>
  </Context>
 </Contract>
 <Parameter>
  <Name>CarBroken </Name>
  <DataType>bool </DataType>
  <DefaultValue ></DefaultValue>
 </Parameter>
 <Parameter>
  <Name>Deposit </Name>
  <DataType>double </DataType>
  <DefaultValue ></DefaultValue>
 </Parameter>
 <Parameter>
  <Name>CarType </Name>
  <DataType>string </DataType>
  <DefaultValue ></DefaultValue>
 </Parameter>
 <Parameter>
  <Name>HasAppointment </Name>
  <DataType>bool </DataType>
  <DefaultValue ></DefaultValue>
 </Parameter>
 <Parameter>
  <Name>NumOfHours </Name>
  <DataType>int </DataType>
  <DefaultValue ></DefaultValue>
 </Parameter>
</CSDL>
```

9.3.3 Service Query Specification

Below, we present the specification of Query1 presented in Chapter 7. Query1 is a traditional exact match style query. The languages SQL and CSQL are used.

SQL Specification

Below is the SQL specification of Query1.

```
TraditionalServiceQuery Query1{
                    ToyotaVehicle toyotavehicle;
                    CarBroken carBroken;
                    Deposit deposit;
                    CarType carType;
                    HasAppointment hasAppointment;
                    NumOfHours numOfHours;
                    Query1Function query1function;
```

```
        RequiredLegalRule1 requiredlegalrule;
        Query1-NF query1-nf;
        Query1ContextInfo query1contexinfo;
}
Attribute ToyotaVehicle{
        vehicletype.value = toyota;
}
RequiredFunction Query1Function{
        RepairShopDomain reparishopdomain;
        ReserveFunctionality servicefunctionality;
        (Query1Precondition query1precondition, EXACT);
        (Query1Postcondition query1postcondition, EXACT);
}
Precondition Query1Precondition{
        $CarBroken == TRUE$;
        $HasAppointment == FALSE$;
}
Postcondition Query1Postcondition{
        $HasAppointment == TRUE$;
}
RequiredNonFunctional Query1-NF{
        (Query1Safety query1safety, EXACT);
}
Safety Query1Safety{
        Query1Function query1function;
        $maxTime < 10days$;
}
LegalIssue RequiredLegalRule{
        (RequiredDepositRule requireddepositrule, EXACT);
```

```
}
DepositRule RequiredDepositRule{
                    amount 0;
                    cuurency CanadianDollar;
}
ContextInfo Query1ContextInfo1{
                    Query1Location query1location;
                    Query1who query1who;
}
Location Query1Location{
                    Query1City query1city;
                    Query1Suburb query1suburb;
}
Region Query1City{
                    type city;
                    name montreal;
}
Region Query1Suburb{
                    type suburb;
                    name downtown;
}
WhoRequester Query1who{
                    membership CAA;
}
```

CSQL Specification

Below is the CSQL specification of Query1.

```
<?xml version=”1.0” encoding=”UTF−8”?>
<Query−w>
 <RequiredFunction>
  <Precondition>
   <Condition>CarBroken==T &&
   HasAppointment==F</Condition>
   <weight>6</weight>
  </Precondition>
  <Postcondition>
   <Condition>HasAppointment==T</Condition>
   <weight>6</weight>
  </Postcondition>
  <Functionality>ReserveFunctionality
  </Functionality>
  <Domain>RepairShopDomain</Domain>
 </RequiredFunction>
 <RequiredNonFunctional>
  <Safety>
   <maxTime>7days</maxTime>
  </Safety>  <ProviderTrust>
 </RequiredNonFunctional>
 <RequiredLegalIssue>
  <DepositRule>
   <Amount>0</Amount>
   <Currency>dollar</Currency>
   <Rule>NoRule</Rule>
   <Date>2011−07−30</Date>
   <Time>00:00:00</Time>
  </DepositRule>
 </RequiredLegalIssue>
 <RequesterContext>
  <WhoRequester>
   <Membership>CAA</Membership>
   <Region>
    <Type>City</Type>
    <Name>Montreal</Name>
   </Region>
   <Region>
    <Type>Suburb</Type>
    <Name>Downtown</Name>
   </Region>
  </WhoRequester>
 </RequesterContext>
 <AuthenticationCertificate>Certificate
  Type1</AuthenticationCertificate>
</Query−w>
```

9.3.4 Service Plan Specification

In response to the SQL specification the Planning Unit prepares the service lookups. The SUL specifications of the service lookup and the lookup result for Query1 are presented below:

```
ServiceLookup Query1Lookup{
            RepairShopDomain reparishopdomain;
            ReserveFunctionality servicefunctionality;
            Query1Certificate;
}
LookupResult Query1LookupResult{
            Garage1Node garage1node;
            Garage2Node garage2node;
            Garage3Node garage3node;
}
```

The plan result is presented below:

```
Plan Query1Result{
    Query1Match1 query1match1;
}
ServiceType Query1Match1{
    ServiceProviderID garage2;
    Garage2-1 garage2-1;
}
```

9.4 Summary

In this chapter a set of languages necessary to support the development process of a service-oriented application have been discussed. A partial specification for the auto roadside emergency service case study is given. The languages are easy to use, yet they have a formal semantic basis.

Chapter 10

Conclusion and Future Work

This thesis introduced an approach for the development of trustworthy context-dependent service-oriented systems. The results of this thesis will have a positive impact in the way service-oriented applications can be developed. The three main contributions are in the areas *service modeling*, *service composition* and *service provision*. Below we first discuss how the research goals set earlier in Chapter 3 have been met by the results of this thesis, next we give an assessment of the solutions, and finally a brief discussion of future work directions are discussed.

10.1 Meeting the Goals

In providing solutions to the research problems we positioned ourselves with respect to (1) current concerns on lack of right methodologies in SOC, (2) the need for formalism, and (3) the need to be practical.

10.1.1 Service Modeling

The major contribution for this area is the formal *ConfiguredService* model. This contribution has remedied the lack of support for specifying nonfunctional, legal and contextual information formally in a service contract. The solutions to the four research problems raised in Chapter 3 are stated below.

- *Providing support for trustworthiness information:* The solution for this problem is provided in Chapter 4. We introduced a formal service model that considers service trust and provider trust. In service trust we specified safety, security, availability and reliability. In provider trust we specified peer recommendations and recommendations from independent organizations.

- *Binding context to the service contract:* The solution for this problem is provided in Chapter 4. We defined the service contract to include the specification of the context. The context includes the context conditions constraining the service contract.

- *Including the legal rules in service definition:* The solution for this problem is provided in Chapter 4. We included the specification of the legal rules within the contract definition.

- *The need for a service model for trustworthy context-dependent services:* The solution for this problem is provided in Chapter 4. We introduced the structure *ConfiguredService* which bundles a service and contract together. The specification of the service includes functional and nonfunctional properties. The specification of the contract includes trustworthiness, legal and context information.

To strengthen claims on completeness, consistency, and correctness we discussed three types of analysis on *ConfiguredService* model in Chapter 4. The *ConfiguredService* and its three stage analysis have been formalized, and can be realized in an implementation.

10.1.2 Service Composition

A major contribution is the new formal composition theory for *ConfiguredService* models. This contribution has remedied the lack of support for including nonfunctional, legal and contextual information in a composition and further in formally checking that certain desirable properties are not violated during composition process. There are 2 research problems stated for this goal. The research problems and the solution provided by this thesis are stated bellow.

- *Lack of a composition theory:* The solution for this problem is provided in Chapter 5. We defined composition operators and their semantics. Based on the semantics a formal composition theory to compose *ConfiguredServices* was given. The composition theory composes all elements of the *ConfiguredService* including trustworthiness properties, legal rules and context information.

- *Formal verification approach:* The solution for this problem is provided in Chapter 6. We defined a formal verification approach that uses model transformation to transform service compositions to timed automata. The timed automata can then be formally verified using the model checking tool UPPAAL.

10.1.3 Service Provision

The main contribution for this area is the service provision framework *FrSeC*. The framework elements and their roles have been formally specified. We also introduced a set of languages to support the processing of *FrSeC*. A blueprint for the required tools to support *FrSeC* has also been proposed. This remedies many inadequacies in current service-oriented computing frameworks. In particular, ranking, dynamic composition of complex queries, and context-dependent trustworthy delivery are our significant contributions. The research problems and the solutions provided by this thesis are stated bellow.

- *Designing a formal service provision framework:* The solutions provided for this problem are in Chapter 7, and in Chapter 8. In the former we introduced *FrSeC*, a formal description of its essential elements, and the communication structure between the elements. In the latter we introduced the dynamic service composition as required by the complex queries. We defined the three types of dynamic service compositions *template-based*, *semi-automatic*, and *automatic*.

- *Defining Languages:* The research problem stated for this goal is *the need for language support*. The solution for this problem is provided in Chapter 9. In it we defined the languages necessary to specify the components of *FrSeC*. We also defined the languages necessary to define the communication between *FrSeC* components.

Two sets of languages were introduced. The first set is for communication either enabled by or received by humans. The second set is XML-based, and is to be used by computers to define communications.

10.2 Assessment

In this section, we evaluate the *ConfiguredService* model with respect to the following criteria: *completeness*, *comprehensibility*, *modifiability*, *testability*, *reusability*, and *scalability*.

Completeness: *Are the elements of the service model sufficient to model trustworthy context-dependent service-oriented systems?* The following factors support our argument that the elements of the formal model are sufficient to express various trustworthy context-dependent service-oriented systems:

- *Service Functionality*: If the attribute part in the Service part of *ConfiguredService* is sufficient to evaluate the pre and postconditions stated for the functionality of service, then we can say that the service part is complete.

- *Context*: To use context in constraining a contract, a set of context rules should be included in the service definition. It is a shared responsibility of service provider and service requester in making context information sufficient so that (1) service can be selected, and (2) the service can be delivered.

- *Trustworthiness*: It is the responsibility of the service provider to make trustworthiness claims precise and complete. Incompleteness may result in service requester not knowing enough about the service, and lack of precision might cause the TA in not being able to analyze its truthfulness. Such incompleteness can be spotted during the analysis stage.

- *Expressive Power*: The proposed *ConfiguredService* model has been applied to the Auto Roadside Emergency Service [tBGKM08], [Koc07] and [BK07] problem. The result shows that it is sufficiently expressive to model the service requirements of this example.

Comprehensibility: *Is the description easy to understand?* The *ConfiguredService* is a structured and precise informal description for naive users. Two other representations of it have been provided. One is the *ConfiguredService Specification Language (CSL)* for the use of non-experts in *FrSeS*. The other is the *ConfiguredService Description Language* that is mainly intended to be used by the TA for formal analysis.

Modifiability: *How easy it is to modify the specification?* Every element in our formal model is described separately. For example, a contract is specified separately from a service definition. This enables modifying the contract without affecting the definition of a service.

Testability: *Is it possible to validate whether or not a specification is correct?* In Section 4.3, we have provided the set of properties that can be tested at the different stages of a *ConfiguredService* life. We have introduced how each property can be analyzed.

Reusability: *Does the formal model support reuse?* Since every element in our service model is described separately, it is possible to reuse these definitions across different *ConfiguredServices* in a specific application, as well as across different applications.

Scalability: *Does the formalism scale up to handle large problems?* The composition theory enables the creation of composite services incrementally. This will enable the construction of large service-oriented applications and hence our approach is scalable.

10.3 Future Work

10.3.1 Implementation

Developing trustworthy context-dependent service-oriented applications is a complex process. In addition to the contributions of this thesis, we have identified the set of tools that will facilitate the development activities in *FrSeC* to become as automatic as possible. Figure 46 shows the set of proposed tools. So far we have implemented CTT. We have also implemented a tool to simulate the behavior of the Planning Unit. This tool is meant to match and rank *ConfiguredServices* according to the matching and ranking algorithms presented in Chapter 7. This tool has been tested on the auto roadside emergency service case study. As part of our intended future work, we intend to define the requirements, challenges and implement the tools introduced below.

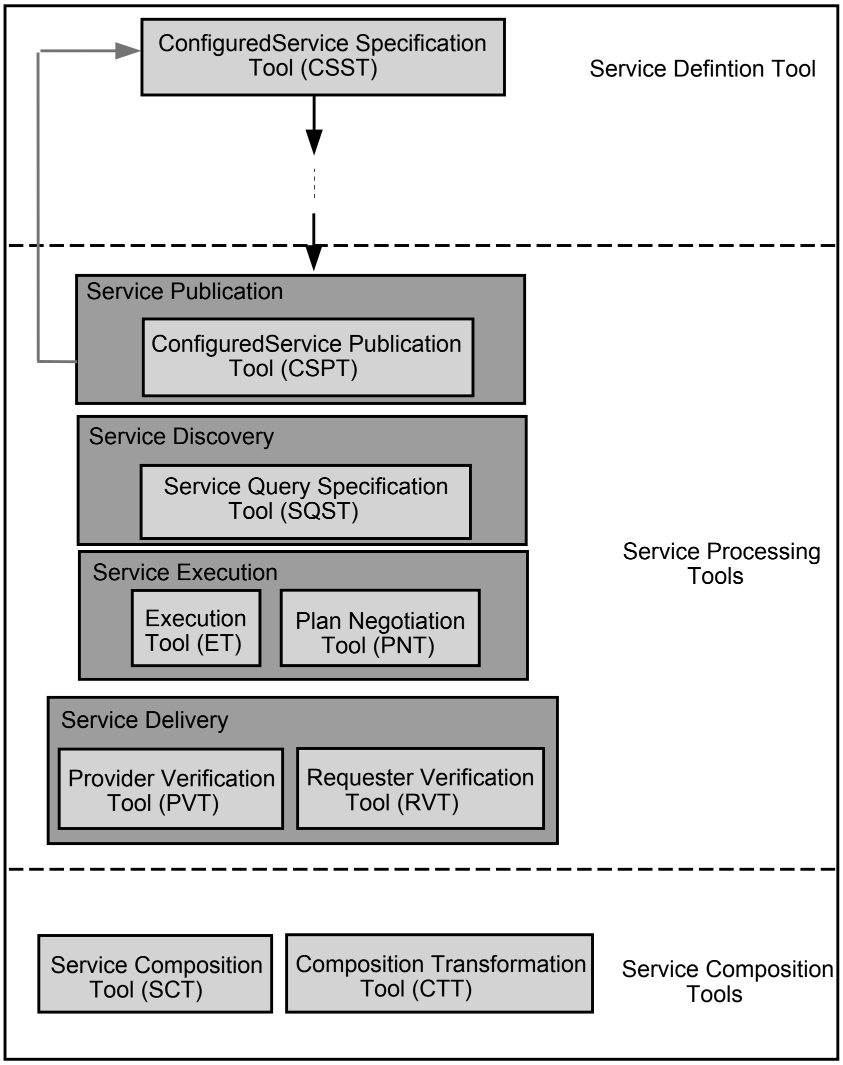

ConfiguredService Specification Tool (CSST)
Service Defintion Tool
Service Publication
ConfiguredService Publication Tool (CSPT)
Service Discovery
Service Query Specification Tool (SQST)
Service Execution
Execution Tool (ET)
Plan Negotiation Tool (PNT)
Service Processing Tools
Service Delivery
Provider Verification Tool (PVT)
Requester Verification Tool (RVT)
Service Composition Tool (SCT)
Composition Transformation Tool (CTT)
Service Composition Tools

Figure 46: Tool Support

Service Definition Tool

Service providers use the *ConfiguredService Specification Tool (CSST)* to generate *ConfiguredServices*. Since *ConfiguredService* description is formal and the service provider may lack a formal background this tool is essential to ensure that only correctly formatted formal descriptions of *ConfiguredServices* are produced. With its graphical user interface, CSST collects the relevant information from a service provider, and assembles the formal structure of the service part and contract part in the *ConfiguredService* service. CSST provides a user friendly interface to model the *ConfiguredServices*. CSST will be designed in such a way that different views of the assembled *ConfiguredService* are projected and the service provider can interactively refine the input until the desired formal document is produced.

Service Processing Tools

These tools are necessary to process service publication, service discovery, service execution, and service delivery.

Service Publication Tool During service publication, services provider use the *ConfiguredService publication tool (CSPT)*, which is a graphic-based tool. The CSPT receives its input in the form of CSL from CSST, produces a certificate request in TAL and interacts with the TA for getting the certificate. CSPT is used by service providers to browse the Service Registry using the certificate received from the TA. The browsing result is sent from the Service Registry in SRL. CSPT is then used by the service provider to submit his *ConfiguredService* to the TA for publication. The transmitted *ConfiguredService* will be in CSDL.

Service Discovery Tools During service discovery, service requesters can use the *Service Query Specification Tool (SQST)* to perform the following tasks:

- send and receive certificate requests to the Trusted Authority in TAL (which is automatically translated to TADL for communication by SQST),

- browse the Service Registry content and receive the result in SRL,
- formulate service queries in SQL (which is automatically translated to CSQL for communication by SQST),
- formulate composition queries in SQL (which is automatically translated to CSQL for communication by SQST),
- refine a service query depending on the Planning Unit feedback which is performed by changing the SQL specification depending on the result of the query received in SUL, and
- formulate negotiation request in NUL (which is automatically translated to NUDL for communication by SQST).

Service Execution Tools During service execution, two tools are necessary to manage execution scenarios and planning. The proposed tools for these purposes are Execution Tool (ET) and the Plan Negotiation Tool (PNT). ET is the run-time environment assistant and it is responsible for automating the execution of plans received from service requesters in SUDL. PNT automates the behavior of the Plan Negotiation Unit. It receives negotiation requests from the service requester in NUDL and communicates with service providers to perform the negotiation. The result of the negotiation is sent back to the service requester in NUDL.

Service Delivery Tools Two tools are suggested for assisting post delivery service actions. The first tool is the *Provider Verification Tool (PVT)*, which is to be used by service providers to verify the satisfaction of the contract during service provision. It enables service providers to submit and communicate with the Trusted Authority to perform after delivery analysis discussed in Section 4.3.3.

The second tool is the *Requester Verification Tool (RVT)*. This tool is to be used by Service Requesters to verify the satisfaction of the contract after service provision. It also verifies the provision of the postconditions and output parameters. This tool is connected to the Trusted Authority to enable Service Requesters perform the after delivery analysis discussed in Section 4.3.3.

The main difference between PVT and RVT is that they are intended for different user communities, and their respective analyses are as discussed in Section 4.3.3.

Service Composition Tools

Two tools are required for automating service compositions. The first tool is the *Service Composition Tool (SCT)*. It can be used by service providers to perform static service compositions, either graphically or using SUL. It enables the access to the Registry to get information about available domains, functionalities and *ConfiguredServices*. SCT also verifies the syntactic correctness of service compositions. It will enable service providers to submit service compositions to the TA for analysis.

The second tool is the *Composition Transformation Tool (CTT)*. It is responsible for transforming service compositions defined in SUL into models understood by model checking tools such as UPPAAL. It automatically performs the transformation and verifies the correctness and completeness of it. This tool is mainly used by the TA. Service compositions submitted by service providers to the TA are verified with the help of this tool.

10.3.2 Extending FrSeC to the Cloud

Cloud computing is any IT resource that exists outside of an enterprise firewall that may be leveraged by an enterprise over the Internet. These resources may include storage, database, application development, and application services. The main motivation behind cloud computing is that it is cheaper to leverage these resources as services, paying as you go as you need them, than it is to buy more hardware and software for the data center.

As part of our future work, we intend to use the capabilities of the cloud to support *FrSeC*. Below is a brief discussion of how cloud computing can impact the implementation and use of *FrSeC*.

Service Provider (SP) SP is the main element of *FrSeC* that can make use of the cloud. As discussed earlier service providers define services and publish their associated *ConfiguredServices*. In typical SOA based systems, the services are deployed and executed at the service provider premise. This will definitely restrict service providers. The restriction is mainly associated with available infrastructure. Another set of restrictions are related to location issues. All these restrictions can be tackled using the cloud. The cloud provides unlimited infrastructure. The service provider can deploy an instance of the service in the cloud in India and another instance in Canada.

Execution Unit (EU) EU in *FrSeC* is responsible for monitoring service executions and executing service compositions. It can make big use of the cloud. Multiple instances of EU can be deployed in the cloud. An instance or more can be assigned to cover each area or zone. Some instances can be assigned to special service requesters. The cloud will increase the response time, reliability and availability of EU.

Service Registry (SRe) In *FrSeC*, there is a centralized service registry. The use of the cloud will enable us to have multiple instances of the same registry all over the cloud. The main issue will be ensuring the synchronization between the different instances.

Planning Unit (PU) and Plan Negotiation Unit (PNU) PU and PNU can also be deployed using the cloud. Multiple instances can be deployed in different regions. Specific PU's with special algorithms and special requirements can be assigned to special service requesters. In essence, almost all parts of *FrSeC* provide services. So why not make use of the cloud to provide these services.

10.3.3 Policy Language

In Chapter 4, we have introduced the formal service model *ConfiguredService*. As part of a *ConfiguredService* contract the security policies guaranteed by the service are stated. Currently, these policies are specified as strings. As part of our future work, we intend to define a new policy language for the specification of context-dependent security policies.

10.3.4 CSDL-to-WSDL Transformation

The *Web Services Description Language (WSDL)* [Pap08] is an XML-based language for describing services as collections of communicating endpoints capable of exchanging messages. WSDL is the de facto standard for the specification of Web services. It is supported by many tools and frameworks. As discussed earlier, WSDL focuses only on the specification of the service functionality. On the other hand, *ConfiguredService* is a much richer model. We have defined a set of transformation rules that transform a *ConfiguredService* defined in CSDL into a Web service defined in WSDL. This set of transformation rules is necessary at this stage to ensure the backward compatibility with current standards. As part of our future work, we intend to implement a tool that automates the transformation process. This will ensure that rich services defined as *ConfiguredServices* can still be used using Web services standards and frameworks.

Bibliography

[AES06] Atif Alamri, Mohamad Eid, and Abdulmotaleb El Saddik. Classification of the stat-of-the-art dynamic web services composition techniques. *International Journal of Web and Grid Services*, 2(2):148–166, 2006.

[AGG$^+$05] José Luis Ambite, Genevieve Giuliano, Peter Gordon, Qisheng Pan, Naqeeb Abbasi, LanLan Wang, and Matthew Weathers. Argos: dynamic composition of web services for goods movement analysis and planning. In *Proceedings of the 2005 national conference on Digital government research (dg.o 2005)*, pages 275–276. Digital Government Society of North America, 2005.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[And04] A.H. Anderson. An introduction to the web services policy language (wspl). In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 189–, Washington, DC, USA, 2004. IEEE Computer Society.

[AVMM04] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint driven web service composition in meteor-s. In *SCC '04: Proceedings of the 2004 IEEE International Conference on Services Computing*, pages 23–30, Washington, DC, USA, 2004. IEEE Computer Society.

[AW05] José Luis Ambite and Matthew Weathers. Automatic composition of aggregation workflows for transportation modeling. In *Proceedings of the 2005 national conference on Digital government research*, pages 41–49. Digital Government Society of North America, 2005.

[BA08] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

[BBN$^+$06] M. Boreale, R. Bruni, R. De Nicola, I. Lanese, M. Loreti, U. Montanari, D. Sangiorgi, and G. Zavattaro. Scc: a service centered calculus. In *Proceedings of WS-FM 2006, 3rd International Workshop on Web Services and Formal Methods, Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[BDL04a] Gerd Behrmann, Alexandre David, and Kim Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 33–35. Springer Berlin / Heidelberg, 2004.

[BDL04b] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, volume LNCS 3185, pages 200–236. Springer–Verlag, September 2004.

[BHL$^+$02] Mark H. Burstein, Jerry R. Hobbs, Ora Lassila, David Martin, Drew V. McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne, and Katia P. Sycara. Daml-s: Web service description for the semantic web. In *Proceedings of the First International Semantic Web Conference on The Semantic Web (ISWC '02)*, pages 348–363, London, UK, 2002. Springer-Verlag.

[BK07] Dominik Berndl and Nora Koch. Sensoria automotive scenario: Illustrating service specification. Technical report, - FAST, No. 2, August 2007.

[BM08] Maria Grazia Buscemi and Ugo Montanari. Open bisimulation for the concurrent constraint pi-calculus. In Sophia Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2008.

[Boo] Orange Book. 1985, http://csrc.nist.gov/publications/history/dod85.pdf, revised department of defense directive, 2002, http://www.dtic.mil/whs/directives/corres/pdf/850001p.pdf.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithims and tools. Report 316, The United Nation University, P.O.Box 305, Macau, September 2004.

[Cha07] David Chappell. Introducing SCA. Open SOA, SCA Resources. Available at http://www.osoa.org/display/Main/SCA+Resources, July 2007.

[CIJ+00] Fabio Casati, Ski Ilnicki, Li-jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 13–31, London, UK, 2000. Springer-Verlag.

[CKM+03] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next step in web services. *Commun. ACM*, 46(10):29–34, 2003.

[CMX08] Xiao-Xia Cao, Huai-Kou Miao, and Qing-Guo Xu. Modeling and refining the service-oriented requirement. In *TASE '08: Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 159–165, Washington, DC, USA, 2008. IEEE Computer Society.

[dA09] Joao Pedro Abril de Abreu. *Modelling Business Conversations in Service Component Architectures*. Phd thesis, University of Leiceste, July 2009.

[DCP+06] Gregorio Dyaz, M. Emilia Cambronero, Juan J. Pardo, Valentynn Valero, and Fernando Cuartero. Automatic generation of correct web services choreographies and orchestrations with model checking techniques. In *International Conference on Internet and Web Applications and Services/Advanced International Conference on Telecommunications, 2006. AICT-ICIW '06.*, pages 186 – 186, feb. 2006.

[Dey01] Anind K. Dey. Understanding and using context. *Personal Ubiquitous Comput.*, 5(1):4–7, 2001.

[DLSZ06] Jin Song Dong, Yang Liu, Jun Sun, and Xian Zhang. Verification of computation orchestration via timed automata. In *ICFEM06*, volume LNCS 4260, pages 226–245. Springer–Verlag, 2006.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[DST+06] Xie Dan, Ying Shi, Zhang Tao, Jia Xiang-Yang, Liang Zao-Qing, and Yao Jun-Feng. An approach for describing soa. In *International Conference on Wireless Communications, Networking and Mobile Computing, WiCOM 2006*, pages 1–4, Sept. 2006.

[EAS08] Mohamad Eid, Atif Alamri, and Abdulmotaleb El Saddik. A reference model for dynamic web service composition systems. *International Journal of Web and Grid Services*, 4(2):149–168, 2008.

[Erl07] Thomas Erl. SOA *Principles of Service Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In *Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM.

[FFK05] Jesús Arias Fisteus, Luis Sánchez Fernández, and Carlos Delgado Kloos. Applying model checking to bpel4ws business collaborations. In *Proceedings of the 2005 ACM symposium on Applied computing (SAC '05)*, pages 826–830, New York, NY, USA, 2005. ACM.

[FK92] David Ferraiolo and Richard Kuhn. Role-based access control. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[FLB06] José Luiz Fiadeiro1, Antónia Lopes, and Laura Bocchi. A formal approach to service component architecture. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *Web Services and Formal Methods. LNCS, vol 4184*, pages 193–213. Springer, Berlin Heidelberg, 2006.

[FS09] Keita Fujii and Tatsuya Suda. Semantics-based context-aware dynamic service composition. *ACM Trans. on Autonomous and Adaptive Systems*, 4(2):1–31, 2009.

[FUMK03] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. In *Proc. of the eighteen* IEEE *international conference on automated software engineerting* ASE03, pages 152–163, 2003.

[FYCL09] Guisheng Fan, Huiqun Yu, Liqiong Chen, and Dongmei Liu. An approach to analyzing dynamic trustworthy service composition. In Asunción Gómez-Pérez, Yong Yu, and Ying Ding, editors, *The Semantic Web, Fourth Asian Conference, ASWC 2009, Shanghai, China, December 6-9, 2009. Proceedings*, volume 5926 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2009.

[GKP$^+$03] Shahram Ghandeharizadeh, Craig Knoblock, Christos Papadopoulos, Cyrus Shahabi, Esam Alwagait, Jose-Luis Ambite, Min Cai, Ching-Chien Chen, Parikshit Pol, Rolfe Schmidt, Saihong Song, Snehal Thakkar, and Runfang

Zhou. Proteus: A system for dynamically composing and intelligently executing web services. In *Proceedings of the 1st International Conference on Web Services*, Las Vegas, NV, USA, June 2003.

[GLG$^+$06] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. Sock: a calculus for service oriented computing. In *Proceedings of the 4th International Conference on Service-Oriented Computing, volume 4294 of LNCS*, pages 327–338, Chicago, IL, USA, December 2006. Springer.

[GP08] Dimitrios Georgakopoulos and Michael P. Papazoglou. *Service-Oriented Computing*. The MIT Press, 2008.

[HB03] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference*, pages 191–200, Darlinghurst, Australia, 2003. Australian Computer Society, Inc.

[HSS05] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In *Proceedings of the International Conference on Business Process Management (BPM2005), volume 3649 of Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, 2005.

[Ibr08] Naseem Ibrahim. Transforming architecture description of component-based systems for formal analysis. Master thesis, Concordia University, December 2008.

[JWY09] Canghong Jin, Minghui Wu, and Jing Ying. A structure-based approach for dynamic services composition. *Journal of Software*, 4(8):891–898, October 2009.

[JYZ$^+$07] Xiangyang Jia, Shi Ying, Tao Zhang, Honghua Cao, and Dan Xie. A new architecture description language for service-oriented architecture. In *Sixth*

International Conference on Grid and Cooperative Computing (GCC 2007), pages 96 –103, Aug. 2007.

[KBM08] Vipul Kashyap, Christoph Bussler, and Matthew Moran. *The Semantic Web, Semantics for Data and Services on the Web*. Springer, 2008.

[KM05] Dominik Kuropka and Harald Meyer. Survey on service composition. Technical report, The Hasso-Plattner-Institute, 2005.

[Koc07] Nora Koch. Sensoria automotive case study: Uml specification of on road assistance scenario. Technical report, - FAST, No. 1, August 2007.

[Kov] Daniel L. Kovacs. Bnf definition of pddl3.1: completely corrected, without comments, unpublished manuscript from the ipc-2011 website, 2011.

[KPP06] Raman Kazhamiakin, Paritosh Pandya, and Marco Pistore. Timed modelling and analysis in web service compositions. In *The First International Conference on Availability, Reliability and Security (ARES 2006)*, page 7, april 2006.

[Lug08] George F. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison-Wesley Publishing Company, USA, 6th edition, 2008.

[MA11] Mubarak Mohammad and Vangalur Alagar. A formal approach for the specification and verification of trustworthy component-based systems. *Journal of Systems and Software*, 84:77–104, January 2011.

[MdVHC02] Craig Mundie, Pierre de Vries, Peter Haynes, and Matt Corwine. Trustworthy computing. Microsoft White Paper, October 2002.

[MGLZ07] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. JOLIE: a java orchestration language interpreter engine. *Electronic Notes in Theoretical Computer Science*, 181:19–33, 2007.

[MKB07] Saayan Mitra, Ratnesh Kumar, and Samik Basu. Automated choreographer synthesis for web services composition using i/o automata. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 364 –371, july 2007.

[Moh09] Mubarak Sami Mohammad. *A Formal Component-based Software Engineering Approach for Developing Trustworthy Systems*. Phd thesis, Concordia University, Montreal, Canada, April 2009.

[MPM$^+$04] David Martin, Massimo Paolucci, Sheila McIlraith, Mark, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara. Bringing semantics to web services: The owl-s approach. In *First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, San Diego, California, USA, July 2004.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I & II. *Information and Computation*, 100(1):1–77, 1992.

[MR09] Jim Marino and Michael Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 2009.

[MSK08] Philip Mayer, Andreas Schroeder, and Nora Koch. Mdd4soa: Model-driven service orchestration. In *EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 203–212, Washington, DC, USA, 2008. IEEE Computer Society.

[NM02] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA, 2002. ACM.

[NMFR09] Azadeh Ghari Neiat, Mehran Mohsenzadeh, Rana Forsati, and Amir Masoud Rahmani. An agent-based semantic web service discovery framework. In *Computer Modeling and Simulation, 2009. ICCMS '09. International Conference on*, pages 194–198, Feb. 2009.

[OEtH05] Justin O'Sullivan, David Edmond, and Arthur H. M. ter Hofstede. Formal description of non-functional service properties. Technical report, FIT-TR-2005-01, Queensland University of Technology, Brisbane, Australia, February 2005.

[OR08] Joseph C. Okika and Anders P. Ravn. Classification of soa contract specification languages. In *IEEE International Conference on Web Services*, pages 433–440, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[Orc] Orc. v 1.1. http://orc.csres.utexas.edu/.

[OVvdA$^+$07] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Science of Computer Programming*, 67(2-3):162–198, 2007.

[Pap08] Michael P. Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, first edition, 2008.

[PBS$^+$09] Massimiliano Di Penta, Leire Bastida, Alberto Sillitti, Luciano Baresi, Neil Maiden, Matteo Melideo, Marcel Tilly, George Spanoudakis, Jesus Gorroogoitia Cruz, John Hutchinson, and Gianluca Ripa. Secse–service centric system engineering: An overview. In Elisabetta Di Nitto, Anne-Marie Sassen, Paolo Traverso, and Arian Zwegers, editors, *At Your Service: Service-Oriented Computing from an EU Perspective*, pages 241–272. The MIT Press, 2009.

[PF02] Shankar R. Ponnekanti and Armando Fox. Sword: A developer toolkit for web service composition. In *Proceedings of the 11th International WWW Conference*, 2002.

[RBHJ06] Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Foundations for web services orchestrations: Functional and qos aspects, jointly. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006)*, pages 309 –316, nov. 2006.

[RKL$^+$05] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubn Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005.

[SBDM02] Quan Z. Sheng, Boualem Benatallah, Marlon Dumas, and Eileen Oi-Yan Mak. Self-serv: a platform for rapid composition of web services in a peer-to-peer environment. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 1051–1054. VLDB Endowment, 2002.

[SBI99] Fred B. Schneider, Steven M. Bellovin, and Alan S. Inouye. Building trustworthy systems: Lessons from the ptn and internet. *IEEE Internet Computing*, 3(6):64–72, 1999.

[Sch00] Karsten Schmidt. LoLA: A Low Level Analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer-Verlag, June 2000.

[soa08] Service oriented architecture modeling language (SOAML) - specification for the UML profile and metamodel for services (UPMS). OMG Submission document: ad/2008-11-01. Available at

http://www.omgwiki.org/SoaML/doku.php?id=specification, November 2008.

[Som07] Ian Sommerville. *Software Engineering*. Addison-Wesley, 8th edition, 2007.

[SPvS09] Eduardo Silva, Lus Ferreira Pires, and Marten van Sinderen. Supporting dynamic service composition at runtime based on end-user requirements. In *Proceedings of the 1st Workshop on User-generated Services (UGS2009) at the 7th International Joint Conference on Service Oriented Computing, (ICSOC 2009)*, Stockholm, Sweden, November 2009.

[SSP04] Evren Sirin, , Evren Sirin, and Bijan Parsia. Planning for semantic web services. In *In Semantic Web Services Workshop at 3rd International Semantic Web Conference*, 2004.

[SWZZ03] Haiyan Sun, Xiaodong Wang, Bin Zhou, and Peng Zou1. Research and implementation of dynamic web services composition. In Xingming Zhou, Stefan Jhnichen, Ming Xu, and Jiannong Cao, editors, *Advanced Parallel Processing Technologies, 5th InternationalWorkshop, APPT 2003*, volume 2834 of *Lecture Notes in Computer Science*, pages 457–466. Springer-Verlag, September 2003.

[tBBG07a] Maurice ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Web service composition approaches: From industrial standards to formal methods. In *Second International Conference on Internet and Web Applications and Services, 2007. ICIW '07*, pages 15 –15, may 2007.

[tBBG07b] Maurice H. ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Formal methods for service composition. *Annals of Mathematics, Computing and Teleinformatics*, 1(5):1–5, 2007.

[tBGKM08] Maurice H. ter Beek, Stefania Gnesi, Nora Koch, and Franco Mazzanti. Formal verification of an automotive scenario in service-oriented computing. In

Proceedings of the 30th international conference on Software engineering, ICSE '08, pages 613–622, New York, NY, USA, 2008. ACM.

[Tie09] Francesco Tiezzi. *Specification and Analysis of Service-Oriented Applications*. Phd thesis, Universit degli Studi di Firenze, Florence, Italy, April 2009.

[TP05] Vladimir Tosic and Bernard Pagurek. On comprehensive contractual descriptions of web services. In *IEEE International Conference on e-Technology, e-Commerce, and e-Services*, pages 444–449, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[VDD$^+$03] Debra VanderMeer, Anindya Datta, Kaushik Dutta, Helen Thomas, Krithi Ramamritham, and Shamkant B. Navathe. Fusion: A system allowing dynamic web service composition and automatic execution. In *Proceedings of the IEEE Int. Conference on E-Commerce Technology*, page 399. IEEE Computer Society, 2003.

[VGS$^+$05] Kunal Verma, Karthik Gomadam, Amit P. Sheth, John A. Miller, and Zixin Wu. The METEOR-S approach for configuring and executing dynamic web processes. Technical report, LSDIS Lab, University of Georgia, Athens, Georgia, 2005.

[VR04] Maja Vukovic and Peter Robinson. Adaptive, planning based, web service composition for context awareness. In *Proceedings of the second international conference on pervasive computing*, 2004.

[WA08a] Kaiyu Wan and Vasu Alagar. A context-aware trust model for service-oriented multi-agent systems. In *Proceedings of 1st International Workshop on Quality-of-Service Concerns in Service Oriented Architectures (QoSC-SOA 2008)*, pages 221–236, Sydney, Australia, 2008. Springer-Verlag.

[WA08b] Kaiyu Wan and Vasu Alagar. An intensional functional model of trust. In Yucel Karabulut, John Mitchell, Peter Herrmann, and Christian Jensen, editors, *Trust Management II*, volume 263 of *IFIP Advances in Information and Communication Technology*, pages 69–85. Springer Boston, 2008.

[Wan06] Kaiyu Wan. *Lucx: Lucid Enriched with Context*. Phd thesis, Concordia University, Montreal, Canada, January 2006.

[WBF^{+}08] Martin Wirsing, Laura Bocchi, Jose Luiz Fiadeiro, Stephen Gilmore, Matthias Hoelzl, Nora Koch, Philip Mayer, Rosario Pugliese, and Andreas Schroeder. Sensoria: Engineering for Service-Oriented Overlay Computers. In Elisabetta di Nitto, Anne-Marie Sassen, Paolo Traverso, and Arian Zwegers, editors, *At Your Service: Service Engineering in the Information Society Technologies Program*. MIT Press, 2008.

[WC] WS-CDL. Web services choreography description language version 1.0. W3C Candidate Recommendation. November, 2005. (http://www.w3.org/TR/ws-cdl-10/).

[WMA09] Kaiyu Wan, Mubarak Muhammad, and Vasu Alagar. A formal model of business application integration from web services. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '09, pages 656–667, Berlin, Heidelberg, 2009. Springer-Verlag.

[WPS^{+}03] Dan Wu, Bijan Parsia, Evren Sirin, James Hendler, Dana Nau, and Dana Nau. Automating daml-s web services composition using shop2. In *Proceedings of 2nd International Semantic Web Conference*, 2003.

[WSD] WSDL. Web services description language 1.1. W3C Note. March, 2001. http://www.w3.org/TR/wsdl.

[YK04] Xiaochuan Yi and K.J. Kochut. A cp-nets-based design and verification framework for web services composition. In *IEEE International Conference on Web Services, 2004. Proceedings*, pages 756 – 760, july 2004.

[YPCG05] Yuhong Yan, Yannick Pencole, Marie-Odile Cordier, and Alban Grastien. Monitoring web service networks in a model-based approach. In *Third IEEE European Conference on Web Services (ECOWS 2005)*, page 12, nov. 2005.

[ZCCK04] Jia Zhang, C.K. Chang, Jen-Yao Chung, and S.W. Kim. Ws-net: a petri-net based specification model for web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 420 – 427, july 2004.

[ZkMM06] Michal Zaremba, Mick kerrigan, Adrian Mocan, and Matt Moran. Web services modeling ontology. In Jorge Cardoso and Amit P. Sheth, editors, *Semantic Web Services, Processes and Applications*, pages 63–87. Springer, 2006.

Appendix A

Service Processing Languages

This appendix presents the *Service Processing Languages (SPL)*. Below is a detailed discussion about all constitutes languages and their syntax.

A.1 Service Registry Language (SRL)

In this section we introduce the *Service Registry Language (SRL)* for specifying the elements of the Service Registry and the *ConfiguredServices*. This language is also used by Service Providers to specify the *ConfiguredService* that they wish to publish.

A.1.1 Registry, Domain and Functionality

Below is the SRL syntax to specify the Registry, domain and functionality elements of the Service Registry. The preconditions and postconditions that are defined inside a functionality represents the preconditions and postconditions that are true for all *ConfiguredServices* defined under this functionality. A domain element has an associated set of nonfunctional properties that are related to this specific domain. The associated nonfunctional properties are defined in terms of their names.

Registry $< name > \{$

  (Attribute $< name >$)*;

(Domain $< Domain - name >$)*;

}

Domain $< name >$ {

(Attribute $< name >$)*;

DomainNonFunctional $< name >$;

(Domain $< Domain - name >$)* ||

(Functionality $< name >$,

(ServiceProviderNode $< name >$)*)*;

}

DomainNonFunctional $< name >$ {

(Attribute $< name >$)*;

$string\ propertyName$;

}

Functionality $< name >$ {

(Attribute $< name >$)*;

(Precondition $< name >$)*;

(Postconditon $< name >$)*;

(Parameter $< name >$)*;

}

ServiceProviderNode $< name >$ {

(Attribute $< name >$)*;

$String\ ServiceProviderID$;

(ConfiguredService $< name >$)*;

}

A.1.2 *ConfiguredService*

The SRL syntax for specifying the *ConfiguredService* is based on the formal model presented in Chapter 4. Figure 47 shows the *ConfiguredService* meta-model used in defining the SRL syntax for specifying *ConfiguredServices*.

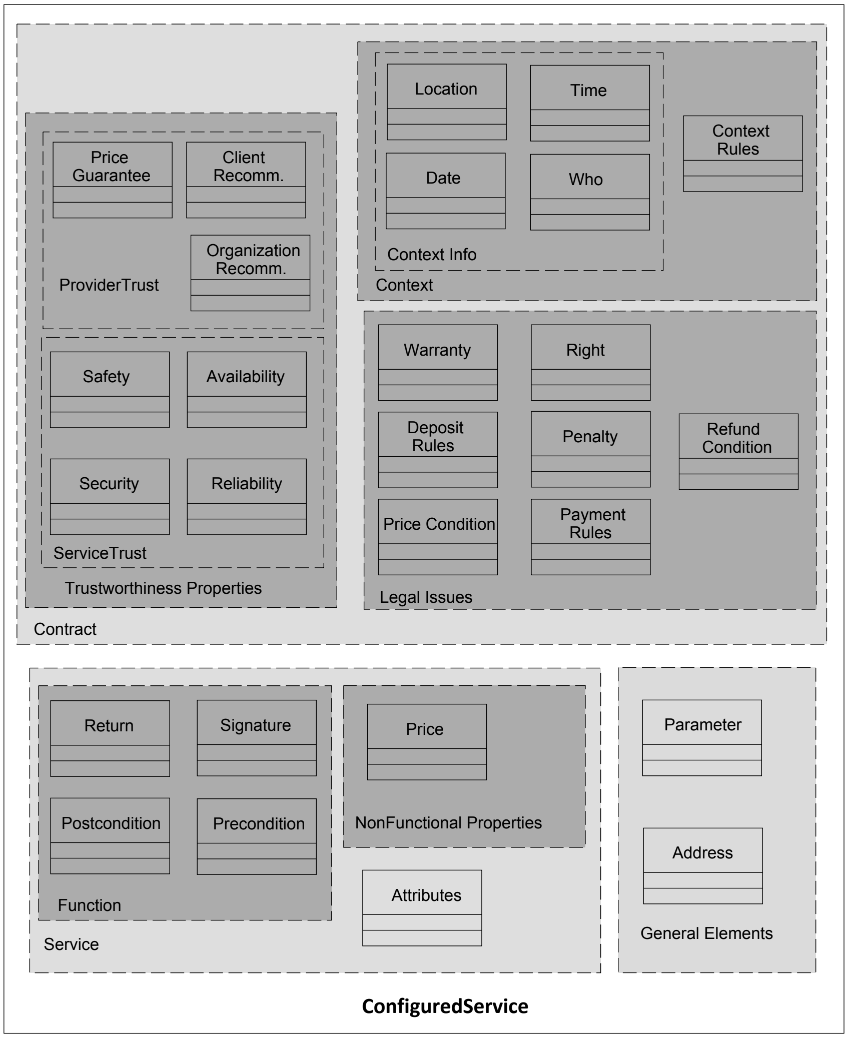


Figure 47: *ConfiguredService* meta-model

The SRL syntax for specifying *ConfiguredService* is presented below. The *ConfiguredService* specification includes the specifications of *DataTypeInfo*, *Attributes*, *Parameters*, *Service* and *Contract*. The specification of DataTypeInfo includes a formal representation of any additional information on data types that can be used for the exchange of services. These data types may be complex abstract data types and DataTypeInfo specification presents a common understating of the structure of these data types.

ConfiguredService $< name >$ {

(DataTypeInfo $< name >$)*;

```
                    (Attribute < $name$ >)*;
                    (Parameter < $name$ >)*;
                    Service < $name$ >;
                    Contract < $name$ >;
}
```

Service The first main part of a *ConfiguredService* is the *service*. In a *service* specification, the functional and nonfunctional properties that *ConfiguredService* guarantee must be formally included. The SRL syntax for specifying a *service* is derived from the formal model in Chapter 4 and *service* specification is presented below.

```
Service < $name$ > {
        (Attribute < $name$ >)*;
        Function < $name$ >;
        NonFunctional < $name$ >;
}
```

Function The *ConfiguredService service* contains the definition of the functional properties guaranteed by this *ConfiguredService*. The SRL syntax for specifying the functional properties part is shown below.

```
Function < $name$ > {
        (Attribute < $name$ >)*;
        Signature < $name$ >;
        Return < $name$ >;
        (Precondition < $FOPL$ >)*;
        (Postconditon < $FOPL$ >)*;
}
```

The function consists of a signature, return information, preconditions and postconditions. The signature consists of the address to invoke this function, the list of parameters associated with this function and the identification information distinguishing this function. The return information defines the return identification and the associated parameters. The preconditions and postconditons are constraints defined in first order predicate logic (FOPL). These definitions conform to the formal definition of a *ConfiguredService* presented in Chapter 4. The SRL specification for specifying the functionality details is presented below.

```
Signature < name > {
        (Attribute < name >)*;
        Address < name >;
        String id;
        (Parameter < name >)*;
}
Result < name > {
        (Attribute < name >)*;
        String id;
        (Parameter < name >)*;
}
Precondition < name > {
        (Attribute < name >)*;
        (Constraint < FOPL >)*;
}
Postcondition < name > {
        (Attribute < name >)*;
        (Constraint < FOPL >)*;
}
Address < name > {
        String url;
}
```

Nonfunctional A *ConfiguredService* definition includes the nonfunctional properties that it can guarantee. Each nonfunctional property is associated with a specific function. SRL syntax is sufficiently expressive to specify any nonfunctional property. Below, the SRL syntax to specify price is shown.

Price is an important nonfunctional property as it plays a major role is service selection. Price defines the cost of invoking a service functionality. It is associated with a currency, a validity duration, conditions if any on the price, discounts if any, and unit of pricing. Examples of unit of pricing are price per use, price per day or price per byte.

```
NonFunctional < name > {
                (Attribute < name >)*;
                (Price < name >)*;
}
Price < name > {
      (Attribute < name >)*;
      double value;
      string currency;
      Duration < priceValidity >;
      PricingUnit < unit >;
      PriceCondition < FOPL >;
      Discount < name >;
      Function < name >;
}
Discount < name > {
          (Attribute < name >)*;
          double value;
          DiscountCondition < FOPL >;
}
```

Contract The second part of a *ConfiguredSerivce* specification is a *contract*. The information in the *contract* constrains the provision of the *ConfiguredSerivce service*. The information defined in the *service* are usually static. While the information defined in the contract are dynamic. A *contract* includes the main parts *trustworthiness*, *legal rules* and *context*. Below is the SRL for specifying a *ConfiguredService contract*.

```
Contract < name > {
        (Attribute < name >)*;
        Trustworthiness < name >*;
        (LegalIssue < name >)*;
        Context < name >;
}
```

Trustworthiness *Trustworthiness* is the system property that denotes the degree of user confidence that the system will behave as expected. In SRL, trustworthiness is defined into *ServiceTrust* and *ProviderTrust*. Below is the SRL for specifying trustworthiness.

```
Trustworthiness < name > {
                (Attribute < name >)*;
                ServiceTrust < name >;
                ProviderTrust < name >;
}
```

ServiceTrust defines the trustworthiness properties that are related to service provision. It includes the features safety, security, availability and reliability. Below is the SRL specification for specifying *ServiceTrust*.

```
ServiceTrust < name > {
            (Attribute < name >)*;
            Safety < name >;
            Security < name >;
            Availability < name >;
            Reliability $< name >$;
}
Safety $< name >$ {
      (Attribute $< name >$)*;
      (Constraint $< FOPL >$)*;
      $float\ maxTime$;
}
Availability $< name >$ {
          (Attribute $< name >$)*;
          (Constraint $< FOPL >$)*;
          $float$ availabilityRate;
}
Reliability $< name >$ {
          (Attribute $< name >$)*;
          (Constraint$< FOPL >$)*;
          $float$ reliabilityValue;
}
Security $< name >$ {
        (Attribute $< name >$)*;
        DateIntegrity $< name >$;
        Confidentiality $< name >$;
}
DataIntegrity $< name >$ {
            (Attribute $< name >$)*;
            (Protocol $< name >$)*;
}
Confidentiality $< name >$ {
             (Attribute $< name >$)*;
             (Protocol $< name >$)*;}
```

ProviderTrust defines the trustworthiness properties that are related to the service provider. A service provider should include, as part of service publication, information on 'dependability issues' that are taken into account during the design and development of the services. This information is a 'seal of trust' of the service provider that may be verified by the Service Requesters when they obtain their services. Any failure in satisfying the 'seal of trust' will lower the 'trust' level of the service provider. Thus, 'seal of trust' is a 'peer-to-peer' trust information. Provider trust may also include 'third party' information that might increase the level of trust that Service Requesters have in a service provider. This information normally includes recommendations from other clients, lowest prices guarantees, payment security guarantees and recommendations from other independent organizations. A buyer may trust a seller because that the seller has been dealt with before, or the seller is recommended by a trusted friend, or the seller is associated with a certain organization or board. There is no agreed upon definition for ProviderTrust. The main issue here is the inclusion of verifiable information that makes a seller trusted. Below is the SRL for specifying provider trust.

```
ProviderTrust < name > {
            (Attribute < name >)*;
            (Recommendation < name >)*;
            PaymentSecurity < name >;
            (ClientEndorsement < name >)*;
            PriceGuarantee < name >;
}
Recommendation < name > {
                (Attribute < name >)*;
                string recommendationAgency;
                string recommendationData;
}
PaymentSecurity < name > {
                (Attribute < name >)*;
                string securityInfo;
}
ClientEndorsement < name > {
                 (Attribute < name >)*;
                 string clientName;
                 string endorsementDate;
}
PriceGuarantee < name > {
              (Attribute < name >)*;
              Price < name >;
              Condition < FOPL >;
              string guarantee;
}
```

Legal Issues One of the essential elements of the *ConfiguredService* contract is the set of legal rules that constrain the contract. Below is SRL syntax for specifying these legal issues. Currently, SRL supports a predefined set of legal rules, although it can be extended to include other legal rules. The rules currently supported by SRL are inspired by the work presented in [OEtH05].

```
LegalIssue < name > {
          (Attribute < name >)*;
          (PriceCondition < name >)*;
          (RefundCondition < name >)*;
          (JoiningFee < name >)*;
          (InterestCharge < name >)*;
          (AdminstrativeCharge < name >)*;
          (DepositRule < name >)*;
          (PaymentRules < name >)*;
          (RequesterPenalty < name >)*;
```

```
        (ProviderPenalty $< name >$)*;
        (RequesterRights $< name >$)*;}
PriceCondition $< name >$ {
            (Attribute $< name >$)*;
            Price $< name >$;
            (Condition $< FOPL >$)*;
}
RefundCondition $< name >$ {
            (Attribute $< name >$)*;
            Refund $< name >$;
            (Condition $< name >$)*;
}
Refund $< name >$ {
            (Attribute $< name >$)*;
            $double\ amount$;
            $string\ currency$;
}
JoiningFee $< name >$ {
        (Attribute $< name >$)*;
        $double\ amount$;
        $string\ currency$;
}
IntrestCharges $< name >$ {
        (Attribute $< name >$)*;
        $double\ amount$;
        Time $< deadlineTime >$;
        Date $< deadlineDate >$;
}
AdminstrativeCharge $< name >$ {
        (Attribute $< name >$)*;
        $double\ amount$;
        Condition $< FOPL >$;
}
DepositRule $< name >$ {
        (Attribute $< name >$)*;
        $double\ amount$;
        $string\ currency$;
        Time $< depositTime >$;
        Date $< depositDate >$;
}
PaymentMethod $< name >$ {
            (Attribute $< name >$)*;
            $string\ paymentMethod$*;
}
PaymentTime $< name >$ {
          (Attribute $< name >$)*;
          Time $< paymentTime >$;
          Date $< paymentDate >$;
}
PaymentDiscount $< name >$ {
          (Attribute $< name >$)*;
          $double\ amount$;
          Condition $< FOPL >$;
}
PaymentMethodFee $< name >$ {
          (Attribute $< name >$)*;
          $double\ amount$;
          $string\ paymentMethod$;
```

```
}
PreferredPayment < name > {
            (Attribute < name >)*;
            string paymentMethod;
}
PaymentRules < name > {
            (Attribute < name >)*;
            (PaymentMethod < name >)*;
            (PaymentTime < name >)*;
            (PaymentDiscount < name >)*;
            (PaymentMethodFee < name >)*;
            (PreferredPayment < name >)*;
}
RequesterPenalty < name > {
               (Attribute < name >)*;
               (Condition < FOPL >)*;
               double amount;
               string currency;
}
ProviderPenalty < name > {
               (Attribute < name >)*;
               (Condition < FOPL >)*;
               double amount;
               string currency;
}
RequesterRights < name > {
               (Attribute < name >)*;
               Warranty < name >)*;
}
```

```
Warranty < name > {
            (Attribute < name >)*;
            (Condition < FOPL >)*;
            Duration < name >;
}
```

Context A context specification in SRL includes the specification of *ContextInfo* and the specification of *ContextRules*. The ContextInfo part specifies the dimensions and tags, which are the contextual properties of the service. The ContextRules part specifies the contextual conditions that should be true for a service to guarantee a function with its associated nonfunctional guarantees. Each rule in ContextRule is associated with a function. Rules are defined as constraints in first order predicate logic. Below is the SRL syntax for specifying context.

```
Context < name > {
      (Attribute < name >)*;
      (ContextRule < name >)*;
      ContextInfo < name >;
}
ContextRule < name > {
      (Attribute < name >)*;
      Constraint < FOPL >;
      Function < name >;
}
ContextInfo < name > {
      (Attribute < name >)*;
      (Location < name >)*;
      (Time < name >)*;
      (Date < name >)*;
      WhoRequester < name >;
```

```
        WhoProvider < name >;
}
```

We illustrate ContextInfo specification in SRL using the three dimensions *WHERE*, *WHEN* and *WHO*. SRL syntax to specify a context is shown below. The dimension WHERE is associated with a location, and the SRL syntax for WHERE specification is shown below. The following are some ways of defining location information.

- *Point*: A location is modeled by a point (as in GPS) that has a longitude and latitude.
- *Region*: A location can be modeled as a region, which can be either a suburb, or a city, or a country or a continent.
- *Address*: A location can be an address, which includes information on street name, door number, a postal code, a city, a country and a phone number.
- *Route*: A location may be a path, which is a sequence of points.
- *URI*: A location may be a Uniform Resource Identifier (URI), which is a string of characters used to identify a name or a resource on the Internet.
- *IP*: A location can be an Internet Protocol address (IP address), which is a numerical label assigned to each device in a computer network.

```
Location < name > {
        (Attribute < name >)*;
        Point < name >;
        Route < name >;
        Region < name >;
        Address < name >;
        URI < name >;
        IP < name >;
}
```

```
Point $< name >$ {
    (Attribute $< name >$)*;
    Longitude $< name >$;
    Latitude $< name >$;
}
Longitude $< name >$ {
        (Attribute $< name >$)*;
        $int\ degrees$;
        $int\ minutes$;
        $int\ seconds$;
}
Latitude $< name >$ {
      (Attribute $< name >$)*;
      $int\ degrees$;
      $int\ minutes$;
      $int\ seconds$;
}
Route $< name >$ {
     (Attribute $< name >$)*;
     (Point $< name >$)*;
}
Region $< name >$ {
      (Attribute $< name >$)*;
      $string\ type$;
      $string\ name$;
}
Address $< name >$ {
      (Attribute $< name >$)*;
      $string\ streetName$;
      string streetAddress;
      string unitNumber;
      string postalCode;
      Region < city >;
      Region < country >;
      PhoneNumber < name >;
}
PhoneNumber < name > {
            (Attribute < name >)*;
            int countryAreaCode;
            int localAreaCode;
            int phoneNumber;
            int extension;
}
URI < name > {
      (Attribute < name >)*;
      string data;
}
IP < name > {
  (Attribute < name >)*;
  string data;
}
```

The dimension WHEN is associated with time and date information. The time information is specified in terms of seconds, minutes and hours. The date information is defined in terms of year, month and day. Week number and day number in a week can also be used in defining the date. Below is the SRL for specifying Date and Time information.

```
Date < name > {
     (Attribute < name >)*;
```

```
    int year;
    int month;
    int day;
    int weekNumber;
    int dayOfweek;
}
Time < name > {
    (Attribute < name >)*;
    int hour;
    int minute;
    int second;
}
Duration < name > {
        (Attribute < name >)*;
        Time < startTime >;
        Time < endTime >;
        Date < startDate >;
        Date < endDate >;
}
```

The dimension WHO, as seen below, is associated with Service Providers and Service Requesters. We can also use WHO dimension to associate information from job titles (roles) and business organizations.

```
WhoRequester < name > {
            (Attribute < name >)*;
            string requesterName;
            string consumerName;
            string requesterJob;
            string requesterOrganization;
```

```
          string consumerJob;
          string consumerOrganization;
          string membership;
}
WhoProvider < name > {
        (Attribute < name >)*;
        string providerName;
        string providerJob;
        string providerOrganization;
}
```

A.2 Service Query Language (SQL)

In this section, we present the *Service Query Language (SQL)* in which service queries generated by Service Requesters are specified. SQL is defined by the Planning Unit and it should be used by Service Requesters to define their queries. SQL can be used to specify the two types of query *service query* and *composition query*.

A.2.1 Service Query

As discussed in Chapter 7, *FrSeC* supports *traditional* and *buffet* styles of queries.

Traditional Style

Figure 48 shows the meta-model for traditional style query. This meta-model is the bases for defining the SQL syntax for specifying traditional style query. In traditional style, the query can either be exact match or weighted match. The main difference between the exact match and the weighted match query is the addition of the weights to the query requirements. The SQL syntax for specifying traditional style query is presented below. The traditional match query consists of a set of parameters, a required function, a set of required

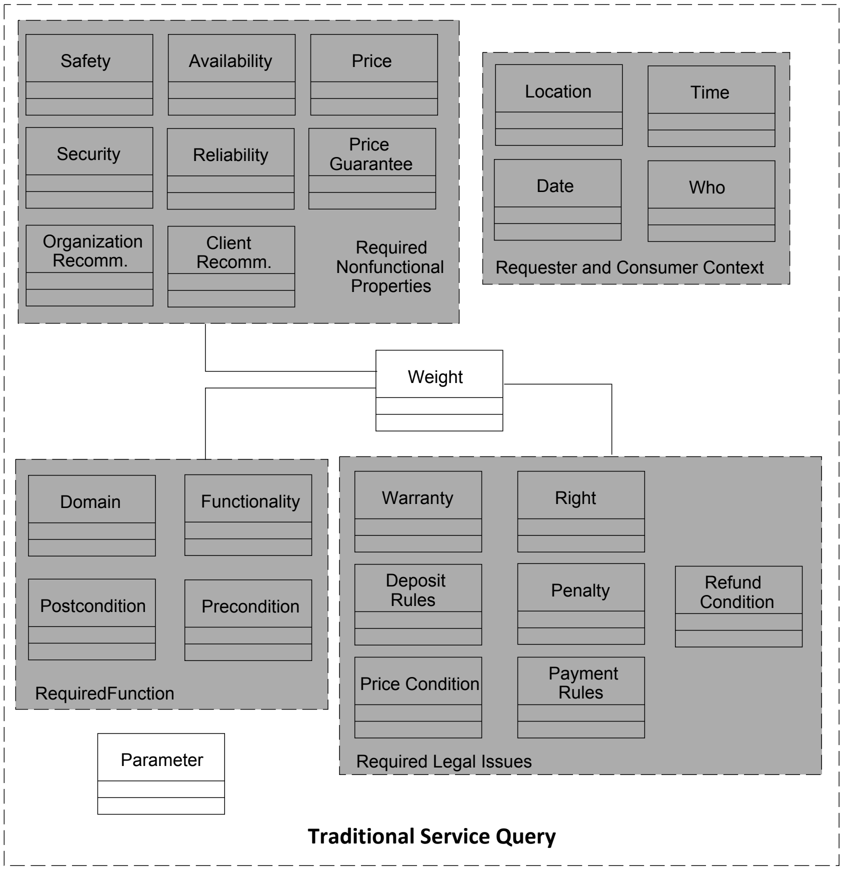


Figure 48: Traditional Query meta-model

nonfunctional properties, a set of required legal rules, requester context information and the service consumer context information. The specification details of those elements are identical to the SRL syntax presented earlier. Each property is associated with a weight. If the query is used to initiated an exact match query all weights will be assigned to “Exact”. In weighted match, the requester is able to assign different weights to different properties.

TraditionalServiceQuery $< name >$ {

(Attribute $< name >$)*;

(Parameter $< name >$)*;

RequiredFunction $< name >$;

(RequiredLegalIssue $< name >$)*;

RequiredNonFunctional $< name >$;

```
                RequesterContextInfo < name >;
                ConsumerContextInfo < name >;
                Authentication Certificate;
}
RequiredFunction < name > {
                (Attribute< name >)*;
                Domain < name >;
                Functionality < name >;
                (Precondition < FOPL >, weight)*;
                (Postconditon < FOPL >, weight)*;
}
ConsumerContextInfo < name > {
                (Attribute< name >)*;
                ContextInfo < name >;
}
RequesterContextInfo < name > {
                (Attribute< name >)*;
                ContextInfo < name >;
}
RequiredLegalIssue < name > {
             (Attribute < name >)*;
             (PriceCondition < name >, weight)*;
             (RefundCondition < name >, weight)*;
             (JoiningFee < name >, weight)*;
             (IntrestCharge < name >, weight)*;
             (AdminstrativeCharge < name >, weight)*;
             (DepositRule < name >, weight)*;
             (PaymentRules < name >, weight)*;
             (RequesterPenalty < name >, weight)*;
```

(ProviderPenalty $< name >$, $weight$)*;

(RequesterRights $< name >$, $weight$)*;

}

RequiredNonFunctional $< name >$ {

(Attribute $< name >$)*;

(Price $< name >$, $weight$);

(Safety $< name >$;

(Security $< name >$, $weight$);

(Availability $< name >$, $weight$);

(Reliability $< name >$, $weight$);

(ProviderTrust $< name >$, $weight$);

}

Buffet Query In the buffet query, the query is defined in terms of specific *ConfiguredServices*. The SQL syntax for specifying buffet style queries is presented below.

BuffetServiceQuery $< name >$ {

(Attribute$< name >$)*;

(Parameter $< name >$)*;

ConfiguredService $< name >$;

RequesterContextInfo $< name >$;

ConsumerContextInfo $< name >$;

$Authentication\ Certificate$;

}

A.2.2 Composition Query

Service composition is of two types static and dynamic. Static service composition is performed by service providers. The result are regular *ConfiguredService* that is seen be service requesters as atomic service. Hence, no special service queries are required. On the other hand, dynamic service composition is driven by user requirements and special types of queries are required. Below we discuss the SQL syntax for specifying dynamic composition queries according to the dynamic composition type.

Template-based The template-based composition query can be define in SQL as following, where $construct$ is one of composition constructs defined in Chapter 5 and $Traditi onalServiceQuery$ is defined above.

CompositionQuery $< name >$ {

(TaditionalServiceQuery$< name >$ **construct** TraditionalServiceQuery$< name >$)*;

}

Semi-automatic The semi-automatic query is initiated by the service requester. In this query, the service requester specifies that he is requesting a semi-automatic service composition and the first required functionality. The functionality is associated with the non-functional requirements, legal requirements and contextual information. In other words, the semi-automatic query will consist of a traditional style associated with information indicating this is not a single traditional query but rather a semi-automatic query. The query is defined in SQL as following:

Semi-AutomaticQueryInitialization $< name >$ {

Name $String$;

TraditionalServiceQuery$< name >$;

}

The planning unit will respond with a set of ranked candidate *ConfiguredServices*. The service requester will respond with the selected *ConfiguredService* and a new service query. The requester second response syntax is defined in SQL below.

Semi-AutomaticQuery $< name >$ {

```
    Name String;
    ConfiguredService < name >;
    TraditionalServiceQuery< name > ;
}
```

The “Name” parameter is used is all interactions corresponding to the same semi-automatic query to differentiate the query from other queries. This is done because the service requester might initiate multiple semi-automatic queries at the same time.

Automatic In automatic query the service requester does not know if the response is a composition or a single service. Hence, no special syntax is necessary. The service requester will just define a traditional style query using the syntax defined above while specifying multiple domains and functionalities.

A.3 Trusted Authority Language (TAL)

TAL is defined by the Trusted Authority. All Service Requesters and Providers should use TAL syntax when defining their certificate requests. Below is the TAL certificate request syntax. To define the legal information we propose to use policies similar to the work in [And04]. The details of this policy language are outside the scope of this thesis. Context information syntax is identical to the definition presented as part of the SRL.

```
CertificateRequest < name > {
                    (Attribute < name >)*;
                    (LegalInformation < name >)*;
                    ContextInfo < name >;
}
LegalInformation < name > {
                    (Attribute < name >)*;
```

```
                (Policy < name >)*;
}
```

TAL can also be used by Service Requesters and Providers for the definition of *ConfiguredService* analysis request sent to the TA. The TAL syntax for analysis requests is presented below.

```
AnalyzeCS < name > {
          (Attribute < name >)*;
          ConfiguredService < name >;
          Property < name >;
}
Property < name > {
        (Attribute < name >)*;
        RequiredValue value;
}
```

TAL is also used by Service Providers and the Planning Unit for defining composition analysis requests. The TAL syntax for specifying composition analysis requests is presented below. Where $plan$ is defined as in Section A.4

```
AnalyzeCompostion < name > {
                 (Attribute < name >)*;
                 Plan < name >;
                 Property < name >;
}
```

A.4 Service Planning Unit Language (SUL)

SUL is used by the Planning Unit to define service lookups, formulate lookup results and service plans. Below is the syntax for the service lookup. A service lookup contains a domain and a functionality.

ServiceLookup $< name >$ {

(Attribute $< name >$)*;

Domain $< name >$;

Functionality $< name >$;

$Authentication\ Certificate$;

}

The syntax of the lookup result is presented below. The lookup result contains a set of nodes. Each node contains a *ConfiguredService* and a service provider ID. Each node satisfies the required domain and functionality defined as part of the lookup.

LookupResult $< name >$ {

(Attribute $< name >$)*;

(ServiceProviderNode $< name >$)*;

}

ServiceProviderNode $< name >$ {

(Attribute $< name >$)*;

$String\ ServiceProviderID$;

(ConfiguredService $< name >$)*;

}

Below is the SUL syntax for specifying a plan. It consists of the specifications of the two parts *ServiceType* and *feedback*. The part *ServiceType* specifies the *ConfiguredServices* and their associated Service Providers. A plan can contain multiple *ServiceTypes* in case of a composition. The relationship between the service types are defined using the constructs defined in Chapter 5. The *feedback* part specifies a feedback, in case of a lack of a match. We use *String* type to specify a feedback. An example of a feedback is *no matches because safety requirement could not be matched*.

Plan $< name >$ {

(Attribute $< name >$)*;

```
    (ServiceType < name > construct ServiceType < name >)*;
    Feedback < name >;
}
ServiceType < name > {
    (Attribute < name >)*;
    (ConfiguredService < name >)*;
}
Feedback < name > {
    (Attribute < name >)*;
    String feedback;
}
```

A.5 Service Negotiation Unit Language (NUL)

NUL is to be used by service requesters to send a verification to plan request and a negotiation request. It is also used by the Negotiation Unit to send back verification and negotiation results. The NUL syntax for these operations is presented below.

```
VerifyPlan < name > {
        (Attribute < name >)*;
        (ConfiguredService < name >)*;
}
VerifyPlanResult < name > {
        (Attribute < name >)*;
        (ConfiguredService < name >)*;
        (Status)*; }
Negotiate < name > {
        (Attribute < name >)*;
        Plan < name >;
```

(ConfiguredService $< name >$)*;

($PropertyName$)*;

($Changes$)*;

}

NegotiateResult $< name >$ {

(Attribute $< name >$)*;

Plan $< name >$;

(ConfiguredService $< name >$)*;

($PropertyName$)*;

($Changes$)*;

}

Appendix B

ConfiguredService Description Language (CSDL)

In previous chapters, we have defined *ConfiguredService* informally and formally. We have also presented the set of languages SPL. The formal definition can be used by formal experts to specify *CofngiuredServices*. SPL can be used by almost any user. On the other hand, these languages cannot be used for passing *ConfiguredServices* between different elements of *FrSeC*. Hence, we introduce CSDL. CSDL is an XML-based language. It is loyal to the formal definition of *ConfiguredService* presented in Chapter 4. CSDL is intended to be used in the background. The user is not required to worry about writing complex XML definitions. The user specify *ConfiguredServices* using SPL which is automatically translated into CSDL. XML has been used as the De facto language in the Web Services industry. Its main advantages include its simplicity, extendability and wide tools support. The rest of this section introduces CSDL.

Figure 49 shows the structure of CSDL file. The root is the CSDL element which represents a *ConfiguredService*. The two main elements of a *ConfiguredService* are *service* and *contract*. Below is XML schema for defining a *ConfiguredService*.

```
<xs:element name="CSDL">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Sevice" type="Service"/>
```

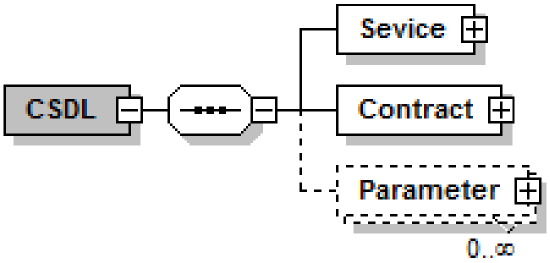


Figure 49: CSDL Root

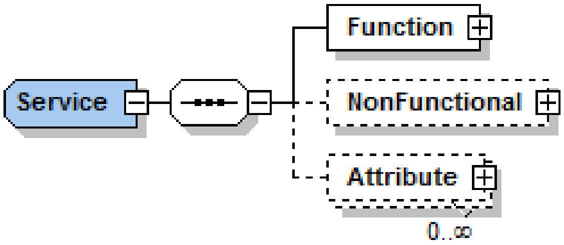


Figure 50: CSDL Service

```
    <xs:element name="Contract" type="Contract"/>
    <xs:element name="Parameter" type="Parameter" minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
```

B.1 Service

Figure 50 shows the structure of the service. The root is the service element. The service includes *function*, *nonfunctional properties* and *attributes*. Below is the XML schema for defining a *ConfiguredService* service.

```
<xs:complexType name="Service">
  <xs:sequence>
    <xs:element name="Function" type="Function"/>
    <xs:element name="NonFunctional" type="NonFunctional" minOccurs="0"/>
    <xs:element name="Attribute" type="Attribute" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
```

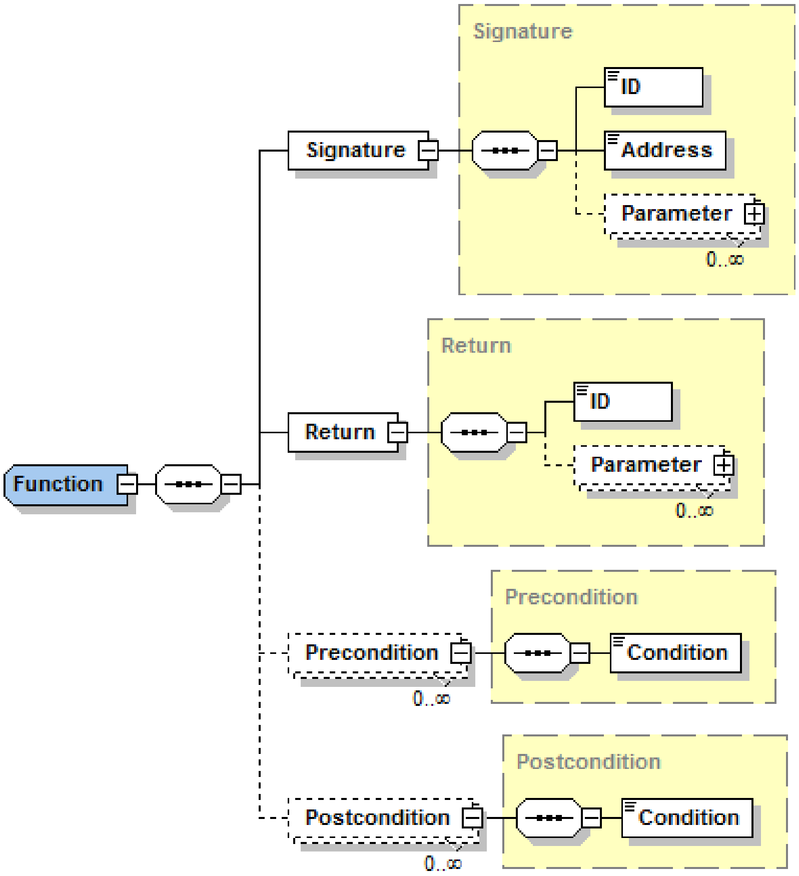


Figure 51: Service Function

B.1.1 Function

The function structure is shown in Figure 51. The function is defined as an XML complexType. The function schema includes the complexTypes *signature*, *result*, *preconditions* and *postconditions*. Below is the XML schema for defining a service function.

```
<xs:complexType name="Precondition">
 <xs:sequence>
  <xs:element name="Condition" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="Postcondition">
 <xs:sequence>
  <xs:element name="Condition" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name=”Signature”>
 <xs:sequence>
  <xs:element name=”ID” type=”xs:string”/>
  <xs:element name=”Address” type=”xs:string”/>
  <xs:element name=”Parameter” type=”Parameter” minOccurs=”0” maxOccurs=”unbounded”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”Parameter”>
 <xs:sequence>
  <xs:element name=”Name” type=”xs:string”/>
  <xs:element name=”DataType” type=”xs:string”/>
  <xs:element name=”DefaultValue” minOccurs=”0”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”Return”>
 <xs:sequence>
  <xs:element name=”ID” type=”xs:string”/>
  <xs:element name=”Parameter” type=”Parameter” minOccurs=”0” maxOccurs=”unbounded”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”Function”>
 <xs:sequence>
  <xs:element name=”Signature” type=”Signature”/>
  <xs:element name=”Return” type=”Return”/>
  <xs:element name=”Precondition” type=”Precondition” minOccurs=”0” maxOccurs=”unbounded”/>
  <xs:element name=”Postcondition” type=”Postcondition” minOccurs=”0” maxOccurs=”unbounded”/>
 </xs:sequence>
</xs:complexType>
```

B.1.2 Nonfunctional properties

Figure 52 shows the structure of the nonfunctional part of a service. Currently, it only includes the complexType price. Price is define as complexType containing value, currency and unit. Below is the XML schema for defining a service nonfunctional property.

```
 <xs:complexType name=”Price”>
  <xs:sequence>
   <xs:element name=”value” type=”xs:double”/>
   <xs:element name=”currecny” type=”xs:string”/>
   <xs:element name=”unit” type=”xs:string”/>
  </xs:sequence>
 </xs:complexType>
```

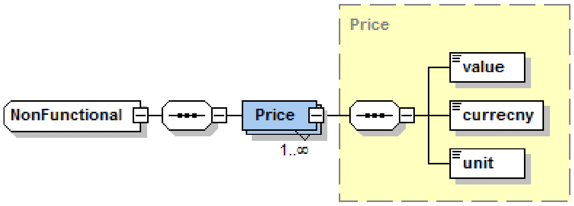


Figure 52: Service NonFunctional

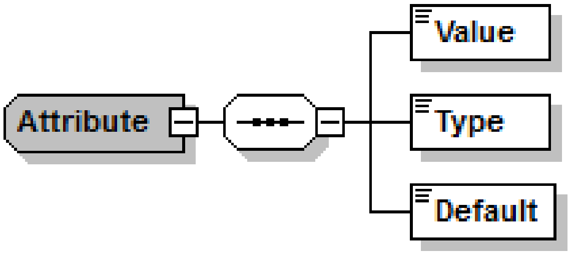


Figure 53: Service Attribute

```
<xs:complexType name=”NonFunctional”>
  <xs:sequence>
    <xs:element name=”Price” type=”Price” maxOccurs=”unbounded”/>
   </xs:sequence>
 </xs:complexType>
```

B.1.3 Attributes

Figure 53 shows the structure of an attribute. It is defined from the simple types value, type and default value. The XML schema for defining an attribute is presented below.

```
<xs:complexType name=”Attribute”>
   <xs:sequence>
     <xs:element name=”Value”/>
     <xs:element name=”Type”/>
     <xs:element name=”Default”/>
    </xs:sequence>
  </xs:complexType>
```

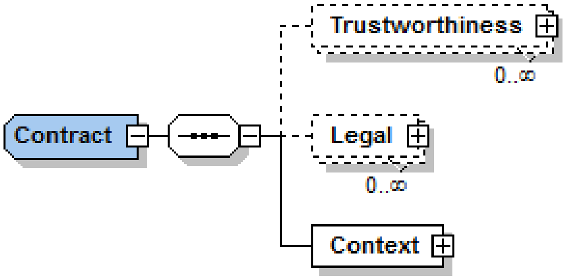


Figure 54: CSDL Contract

B.2 Contract

Figure 54 shows the structure of the contract. The root is the contract element. The contract includes *trustworthiness*, *legal properties* and *context*. The XML schema for defining a contract is presented below.

```
<xs:complexType name="Contract">
  <xs:sequence>
    <xs:element name="Trustworthiness" type="Trustworthiness" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Legal" type="LegalIssue" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="Context" type="Context"/>
   </xs:sequence>
 </xs:complexType>
```

B.2.1 Trustworthiness

Figure 55 shows the structure of the contract trustworthiness part. It includes the two complexTypes *ConfiguredService Trust* and *Provider Trust*. *ConfiguredService Trust* includes the complexTypes safety, security, availability and reliability. *ProviderTrust* include the complexTypes ClientRecommendations, OrganizationalRecommendations and price guarantees. Below is the XML schema for specifying trustworthiness properties.

```
<xs:complexType name="Trustworthiness">
  <xs:sequence>
    <xs:element name="ConfiguredServiceTrust" type="ConfiguredServiceTrust" minOccurs="0"/>
    <xs:element name="ProviderTrust" type="ProviderTrust" minOccurs="0"/>
   </xs:sequence>
 </xs:complexType>
```

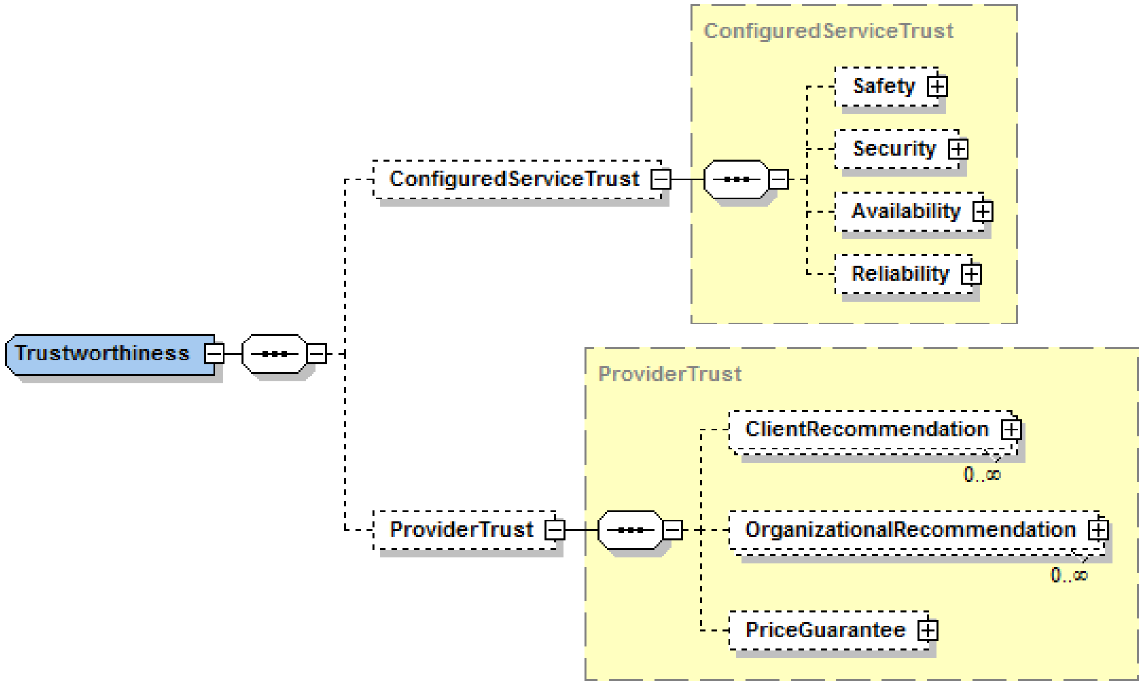


Figure 55: Contract Trustworthiness

```
<xs:complexType name="ConfiguredServiceTrust">
 <xs:sequence>
  <xs:element name="Safety" type="Safety" minOccurs="0"/>
  <xs:element name="Security" type="Security" minOccurs="0"/>
  <xs:element name="Availability" type="Availability" minOccurs="0"/>
  <xs:element name="Reliability" type="Reliability" minOccurs="0"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="Safety">
 <xs:sequence>
  <xs:element name="constraint" minOccurs="0"/>
  <xs:element name="maxTime" type="xs:double" minOccurs="0"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="Security">
 <xs:sequence>
  <xs:element name="DataIntegrityRule" minOccurs="0" maxOccurs="unbounded">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="Rule" type="xs:string"/>
     <xs:element name="weight" type="xs:int"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name=”ConfidentialityRule” minOccurs=”0” maxOccurs=”unbounded”>
   <xs:complexType>
    <xs:sequence>
     <xs:element name=”Rule” type=”xs:string”/>
     <xs:element name=”weight” type=”xs:int”/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”Reliability”>
 <xs:sequence>
  <xs:element name=”constraint” minOccurs=”0”/>
  <xs:element name=”reliabilityRate” type=”xs:double” minOccurs=”0”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”Availability”>
 <xs:sequence>
  <xs:element name=”constraint” minOccurs=”0”/>
  <xs:element name=”availabilityRate” type=”xs:double” minOccurs=”0”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”ClientRecommendation”>
 <xs:sequence>
  <xs:element name=”Client” type=”xs:string”/>
  <xs:element name=”Recommendation” type=”xs:double”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”OrganizationalRecommendation”>
 <xs:sequence>
  <xs:element name=”Organization” type=”xs:string”/>
  <xs:element name=”Recommendation” type=”xs:double”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”PriceGuarantee”>
 <xs:sequence>
  <xs:element name=”Price” type=”Price”/>
  <xs:element name=”Guarantee” type=”xs:boolean”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”ProviderTrust”>
 <xs:sequence>
```

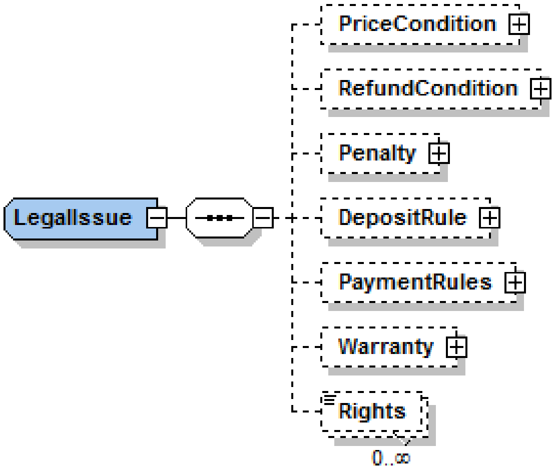


Figure 56: Contract Legal Issues

```
    <xs:element name="ClientRecommendation" type="ClientRecommendation" minOccurs="0"
      maxOccurs="unbounded"/>
    <xs:element name="OrganizationalRecommendation" type="OrganizationalRecommendation"
      minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="PriceGuarantee" type="PriceGuarantee" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

B.2.2 Legal Issues

Figure 56 shows the structure of the legal issues part of a contract. It includes the complexTypes *price conditions*, *refund conditions*, *penalty*, *deposit rules*, *payment rules*, *warranty* and *rights*. Below is the XML schema for defining legal issues in CSDL.

```
<xs:complexType name="LegalIssue">
  <xs:sequence>
    <xs:element name="PriceCondition" type="PriceCondition" minOccurs="0"/>
    <xs:element name="RefundCondition" type="RefundCondition" minOccurs="0"/>
    <xs:element name="Penalty" type="Penalty" minOccurs="0"/>
    <xs:element name="DepositRule" type="DepositRule" minOccurs="0"/>
    <xs:element name="PaymentRules" type="PaymentRules" minOccurs="0"/>
    <xs:element name="Warranty" type="Warranty" minOccurs="0"/>
    <xs:element name="Rights" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name=”PriceCondition”>
 <xs:sequence>
  <xs:element name=”Price” type=”Price”/>
  <xs:element name=”Condition” type=”xs:string”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”RefundCondition”>
 <xs:sequence>
  <xs:element name=”RefundAmount” type=”xs:double”/>
  <xs:element name=”Currency” type=”xs:string”/>
  <xs:element name=”Condition” type=”xs:string”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”Penalty”>
 <xs:sequence>
  <xs:element name=”Amount” type=”xs:double”/>
  <xs:element name=”Currency” type=”xs:string”/>
  <xs:element name=”Condition” type=”xs:string”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”DepositRule”>
 <xs:sequence>
  <xs:element name=”Amount” type=”xs:double”/>
  <xs:element name=”Currency” type=”xs:string”/>
  <xs:element name=”Rule” type=”xs:string”/>
  <xs:element name=”Date” type=”xs:date”/>
  <xs:element name=”Time” type=”xs:time”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”PaymentRules”>
 <xs:sequence>
  <xs:element name=”PaymentTime” type=”PaymentTime”/>
  <xs:element name=”PaymentMethod” type=”PaymentMethod” maxOccurs=”unbounded”/>
  <xs:element name=”PaymentDiscount” type=”PaymentDiscount” minOccurs=”0”/>
  <xs:element name=”PaymentMethodFee” type=”PaymentMethodFee” minOccurs=”0” maxOccurs=”unbounded”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”PaymentMethod”>
 <xs:sequence>
  <xs:element name=”Method” type=”xs:string”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”PaymentTime”>
 <xs:sequence>
```

```
   <xs:element name=”Time” type=”xs:time”/>
   <xs:element name=”Date” type=”xs:date”/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name=”PaymentDiscount”>
  <xs:sequence>
   <xs:element name=”Amount” type=”xs:double”/>
   <xs:element name=”Condition” type=”xs:string”/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name=”PaymentMethodFee”>
  <xs:sequence>
   <xs:element name=”PaymentMethod” type=”PaymentMethod”/>
   <xs:element name=”Fee” type=”xs:double”/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name=”Warranty”>
  <xs:sequence>
   <xs:element name=”Duration” type=”xs:int”/>
   <xs:element name=”Condition” type=”xs:string” minOccurs=”0”
     maxOccurs=”unbounded”/>
  </xs:sequence>
 </xs:complexType>
```

B.2.3 Context

Figure 57 shows the structure of a contract context. It contains the complexType contextInfo and the simpleType context rules. The XML for specifying a contract context is presented below.

```
<xs:complexType name=”Context”>
  <xs:sequence>
   <xs:element name=”ContextInfo” type=”ContextInfo”/>
   <xs:element name=”ContextRules” minOccurs=”0” maxOccurs=”unbounded”/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name=”ContextInfo”>
  <xs:sequence>
   <xs:element name=”Location” type=”Location” minOccurs=”0” maxOccurs=”unbounded”/>
   <xs:element name=”Time” type=”xs:time” minOccurs=”0”/>
   <xs:element name=”Date” type=”xs:date” minOccurs=”0”/>
   <xs:element name=”WhoProvider” type=”WhoProvider” minOccurs=”0”/>
```

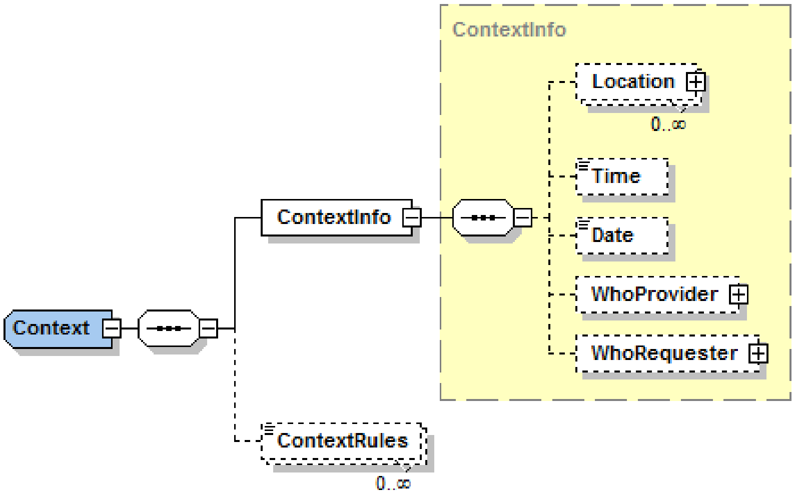


Figure 57: *ConfiguredService* Context

```
    <xs:element name="WhoRequester" type="WhoRequester" minOccurs="0"/>
   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Location">
   <xs:sequence>
    <xs:element name="Point" type="Point" minOccurs="0"/>
    <xs:element name="Route" type="Route" minOccurs="0"/>
    <xs:element name="Region" type="Region" minOccurs="0"/>
    <xs:element name="Address" type="Address" minOccurs="0"/>
    <xs:element name="URI" type="xs:string" minOccurs="0"/>
    <xs:element name="IP" type="xs:string" minOccurs="0"/>
   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Point">
   <xs:sequence>
    <xs:element name="Longitude">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Degrees" type="xs:int"/>
       <xs:element name="Minutes" type="xs:int"/>
       <xs:element name="Seconds" type="xs:int"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="Latitude">
     <xs:complexType>
      <xs:sequence>
     <xs:element name=”Degrees” type=”xs:int”/>
     <xs:element name=”Minutes” type=”xs:int”/>
     <xs:element name=”Seconds” type=”xs:int”/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”Route”>
 <xs:sequence>
  <xs:element name=”Point” type=”Point” maxOccurs=”unbounded”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”Region”>
 <xs:sequence>
  <xs:element name=”Type” type=”xs:string”/>
  <xs:element name=”Name” type=”xs:string”/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”Address”>
 <xs:sequence>
  <xs:element name=”StreetAddress” type=”xs:string” minOccurs=”0”/>
  <xs:element name=”Unit” type=”xs:string” minOccurs=”0”/>
  <xs:element name=”PostalCode” type=”xs:string” minOccurs=”0”/>
  <xs:element name=”Region” type=”Region” minOccurs=”0” maxOccurs=”unbounded”/>
  <xs:element name=”PhoneNumber” minOccurs=”0” maxOccurs=”unbounded”>
   <xs:complexType>
    <xs:sequence>
     <xs:element name=”Number” type=”xs:string”/>
     <xs:element name=”Ext” minOccurs=”0”/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
<xs:complexType name=”WhoRequester”>
 <xs:sequence>
  <xs:element name=”RequesterName” type=”xs:string” minOccurs=”0”/>
  <xs:element name=”ConsumerName” type=”xs:string” minOccurs=”0”/>
  <xs:element name=”RequesterJob” type=”xs:string” minOccurs=”0”/>
  <xs:element name=”RequesterOrganization” type=”xs:string” minOccurs=”0”/>
  <xs:element name=”ConsumerJob” type=”xs:string” minOccurs=”0”/>
  <xs:element name=”ConsumerOrganization” type=”xs:string” minOccurs=”0”/>
  <xs:element name=”Membership” type=”xs:string” minOccurs=”0” maxOccurs=”unbounded”/>
  </xs:sequence>
 </xs:complexType>
<xs:complexType name=”WhoProvider”>
 <xs:sequence>
   <xs:element name=”ProviderName” type=”xs:string” minOccurs=”0”/>
   <xs:element name=”ProviderOrganization” type=”xs:string” minOccurs=”0”/>
   <xs:element name=”ProviderJob” type=”xs:string” minOccurs=”0”/>
  </xs:sequence>
 </xs:complexType>
```

Appendix C

ConfiguredService Query Language

The *ConfiguredService Query Language* is an XML based language used for the specification of service requester requirements. CSQL is to be used for passing service requester queries to the Planning Unit. In this appendix we discuss the CSQL syntax for specifying traditional style queries. Figure 58 shows the structure of the query. It consists of the complexTypes Required Function, Required Nonfunctional, Required Legal Issues, Requester Context, Consumer Context, and the simple type authentication certificate. The XML schema for specifying a traditional query is presented below

```
<xs:element name=”Query–w”>
  <xs:complexType>
   <xs:sequence>
    <xs:element name=”RequiredFunction” type=”RequiredFunction”/>
```

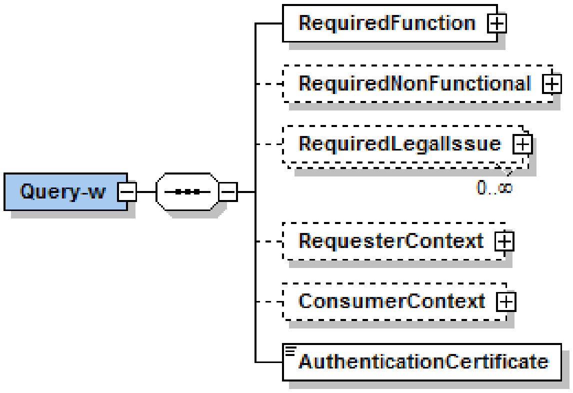


Figure 58: Traditional Query Structure

```
    <xs:element name="RequiredNonFunctional" type="RequiredNonFunctional" minOccurs="0"/>
    <xs:element name="RequiredLegalIssue" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>
      <xs:complexContent>
       <xs:extension base="LegalIssue">
        <xs:sequence>
         <xs:element name="weight" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
       </xs:extension>
      </xs:complexContent>
     </xs:complexType>
    </xs:element>
    <xs:element name="RequesterContext" type="ContextInfo" minOccurs="0"/>
    <xs:element name="ConsumerContext" type="ContextInfo" minOccurs="0"/>
    <xs:element name="AuthenticationCertificate"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
```

C.1 RequiredFunction

Figure 59 shows the structure of a required function. It includes the complexTypes preconditions and postconditions, and the simpleTypes functionality and domain. The XML schema for specifying RequiredFunction is shown below.

```
<xs:complexType name="RequiredFunction">
 <xs:sequence>
  <xs:element name="Precondition" minOccurs="0" maxOccurs="unbounded">
   <xs:complexType>
    <xs:complexContent>
     <xs:extension base="Precondition">
      <xs:sequence>
       <xs:element name="weight" type="xs:int"/>
      </xs:sequence>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>
  </xs:element>
  <xs:element name="Postcondition" minOccurs="0" maxOccurs="unbounded">
   <xs:complexType>
    <xs:complexContent>
```

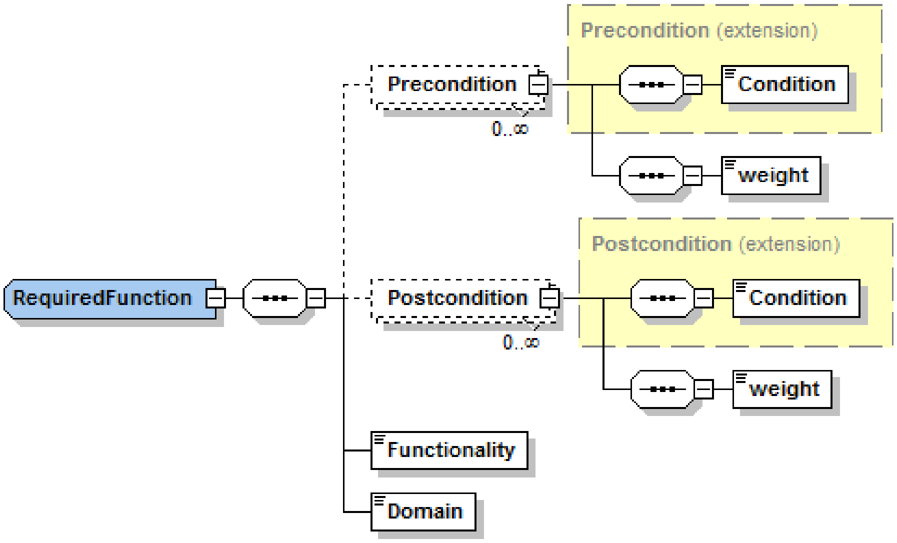


Figure 59: Query RequiredFunction

```
     <xs:extension base=”Postcondition”>
      <xs:sequence>
       <xs:element name=”weight” type=”xs:int”/>
      </xs:sequence>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>
  </xs:element>
  <xs:element name=”Functionality” type=”xs:string”/>
  <xs:element name=”Domain” type=”xs:string”/>
 </xs:sequence>
</xs:complexType>
```

C.2 RequiredNonFunctional

Figure 60 shows the structure of the required nonfunctional properties. The definition of the nonfunctional properties in CSQL is identical to the definition of the nonfunctional properties in CSDL. The only exception is the addition of the weights. Below is the XML schema for defining the required nonfunctional properties in CSQL.

```
<xs:complexType name=”RequiredNonFunctional”>
  <xs:sequence>
```

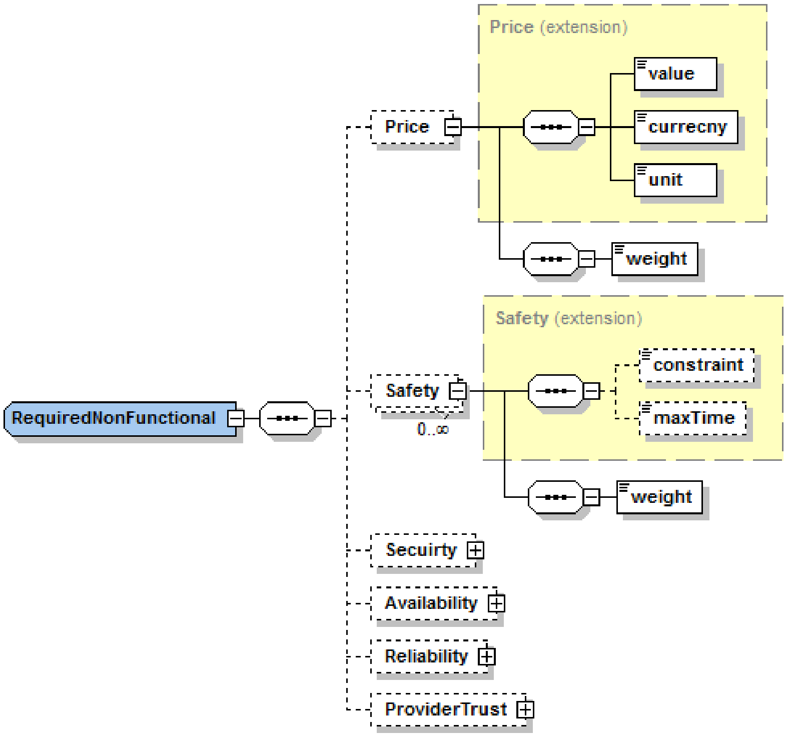


Figure 60: Query RequiredNonFunctional

```
<xs:element name=”Price” minOccurs=”0”>
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base=”Price”>
    <xs:sequence>
     <xs:element name=”weight” type=”xs:int”/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>
<xs:element name=”Safety” minOccurs=”0” maxOccurs=”unbounded”>
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base=”Safety”>
    <xs:sequence>
     <xs:element name=”weight” type=”xs:int”/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
```

```
</xs:element>
<xs:element name=”Secuirty” type=”Security” minOccurs=”0”/>
<xs:element name=”Availability” minOccurs=”0”>
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base=”Availability”>
    <xs:sequence>
     <xs:element name=”weight” type=”xs:int”/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>
<xs:element name=”Reliability” minOccurs=”0”>
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base=”Reliability”>
    <xs:sequence>
     <xs:element name=”weight” type=”xs:int”/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>
<xs:element name=”ProviderTrust” minOccurs=”0”>
 <xs:complexType>
  <xs:sequence>
   <xs:element name=”ClientRecommendation”>
    <xs:complexType>
     <xs:sequence>
      <xs:element name=”weight” type=”xs:int”/>
      <xs:element name=”value” type=”xs:double”/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name=”OrganizationalRecommendation”>
    <xs:complexType>
     <xs:complexContent>
      <xs:extension base=”OrganizationalRecommendation”>
       <xs:sequence>
        <xs:element name=”weight” type=”xs:int”/>
       </xs:sequence>
      </xs:extension>
     </xs:complexContent>
```

```
      </xs:complexType>
     </xs:element>
     <xs:element name=”PriceGurantee”>
      <xs:complexType>
       <xs:complexContent>
        <xs:extension base=”PriceGuarantee”>
         <xs:sequence>
          <xs:element name=”weight” type=”xs:int”/>
         </xs:sequence>
        </xs:extension>
       </xs:complexContent>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
```

C.3 RequiredLegalIssues

The definition of the required legal rules is also identical to the definition of legal rules in CSDL with the addition of the weights. Below is the XML schema for specifying the required legal rules.

```
<xs:element name=”RequiredLegalIssue” minOccurs=”0” maxOccurs=”unbounded”>
     <xs:complexType>
      <xs:complexContent>
       <xs:extension base=”LegalIssue”>
        <xs:sequence>
         <xs:element name=”weight” minOccurs=”0” maxOccurs=”unbounded”/>
        </xs:sequence>
       </xs:extension>
      </xs:complexContent>
     </xs:complexType>
    </xs:element>
```

C.4 Contextual Information

The contract contains the contextual information of the service requester and service consumer. The definition is identical to the CSDL definition of *contextInfo* presented in Chapter B.