WSML Deliverable

D28.1 v0.1
Functional Description of Web Services

WSML Working Draft – January 13, 2006

Authors:
Uwe Keller
Holger Lausen

Editors:
Holger Lausen

Reviewers:
Jos de Bruijn

This version:
http://www.wsmo.org/TR/d28/d28.1/v0.1/20060113/
Latest version:
http://www.wsmo.org/TR/d28/d28.1/v0.1/
Previous version:
http://www.wsmo.org/TR/d28/d28.1/v0.1/20051020/

Abstract

This deliverable is intended to define the semantics of a functional service description, i.e. its capability by means of precondition, assumptions, postconditions and effects. We identify the usage scenarios for functional descriptions and derive requirements on the expressivity of the specification. We briefly overview how existing frameworks for the specification of software components formalize component specifications. Finally, we will define a model-theoretic semantics for capabilities, which will be mapped to WSML in future versions of this deliverable.

Contents

1 Introduction

Software components provide specific and well-defined services to their clients. They are technical means for providing access to services in a well-defined way: clients can use a fixed interface which allows them to interact with the component. By interacting with the component the client is able to consume some specific service. After successful interaction with the component the client will have consumed the intended service and a well-defined state for every party will be reached. Hence, the delivery of a service corresponds to the successful and completed execution of a software component.

A software component does not provide a single service, but instead a class (or set) of distinct and conceptually related (or similiar) services. The precise service to be delivered to clients is determined by the clients themselves: During interaction with the software component they provide certain *input values* to the service which specify the precise details of requested service.

In the simplest case, this is done in a single shot when invoking the component which traditionally can be seen as some sort of procedure call. Using a more sophisticated and general model one can as well assume some more elaborate interaction between the client and the component during which the precise service to be delivered is determined (or negotiated). The interaction between client and component is conceptually (and technically) realized by means of a series of messages which are exchanged. Here, the single messages might be interdependent and the input values can be spread over a sequence of messages. Obviously, the procedure call model is a (very simple) special case of the message-oriented conversation model.

There are two particular aspects of a component which are particularly important for a client and thus should be documented in some way:

- What services can actually be provided by the component? (*Functional Specification*)
- How do I have to interact with the component in a proper way such that I will be able to consume an intended service? (*Behavioural Specification*)

Within this deliverable, we are interested only in specifications of the first type (i.e. functional specification of software components). The most basic and simple way of documentation is the use of natural language prose. The drawback of this sort of documentation is that its meaning easily is ambiguous and not accessible for machines. If one is interested in unambiguous descriptions which are machine processable then formal languages with a mathematically well-defined semantics have to be used.

Enabling automated detection of Web services that adequately serve a given request or usage scenario is a main objective of Semantic Web service technology. Therefore, the functional description of a Web service specifies the provided functionality. Disregarding detailed information on how to invoke and consume the Web service the purpose of functional descriptions is to provide a black box description of *normal runs* of a Web service, i.e. without regard to technical or communication related errors that might occur during service usage.

The contribution of this paper is as follows:

- we present so-called *Abstract State Spaces* as a sufficiently rich, flexible, and language-independent model for describing Web services and the world they act in (Sec. 4.1)

- based on the model we describe what a functional description actually is and properly specify their formal semantics (Sec. 4.1)
- we give concise, formal definitions of all concepts involved in our model (Sec. 4.2)
- we demonstrate the applicability of the introduced model by a specific use case: the semantic analysis of functional descriptions (Sec. 4.3). In particular, we clearly define desirable properties of functional descriptions like realizability and semantic refinement and show how to determine these properties algorithmically based on existing tools in a provably correct way.

Overview of the Solution. As a part of rich model description frameworks like OWL-S and WSMO, functional descriptions of Web services $\mathcal{D}$ are *syntactic expressions* in some specification language $\mathcal{F}$ that is constructed from some (non-logical) signature $\Sigma^{\mathcal{F}}$. Each expression $\mathcal{D} \in \mathcal{F}$ captures specific requirements on Web services W and can be used to constrain the set of all Web services to some subset that is interesting in a particular context. Hence, the set of Web services W that satisfy the functional description D (denoted by $W \models_{\mathcal{F}} \mathcal{D}$) can be considered as actual meaning of $\mathcal{D}$. This way, we can define a natural *model-theoretic semantics* for functional descriptions by defining a satisfaction relation $\models_{\mathcal{F}}$ between Web services and functional descriptions. In comparison to most common logics, our semantic structures (i.e interpretations that are used to assign expressions $\mathcal{D}$ a truth value) are simply a bit more complex. In fact, they can be seen as generalizations of so-called Kripke structures in Modal Logics [Blackburn et al., 2001].

In general, various simpler syntactic elements are combined withing a functional desription $\mathcal{D} \in \mathcal{F}$. State-based frameworks as the ones mentioned above use at least preconditions and postconditions. Whereas $\mathcal{D}$ refers to Web services, these conditions refer to simpler semantic entities, namely *states*, and thus in a sense to a "static" world. Such state conditions ϕ are expressions in (static) language $\mathcal{L}$ over some signature $\Sigma^{\mathcal{L}}$. Single states s determine the truth value of these conditions. Formally, we have a satisfaction relation $\models_{\mathcal{L}}$ between states s and state expressions ϕ, where $s \models_{\mathcal{L}} \phi$ denotes that ϕ holds in state s. In essence, we can observe that on a syntactic level a language $\mathcal{L}$ for capturing static aspects of the world is *extended* to a language $\mathcal{F}$ that captures dynamic aspects of the world.

In order to define a similar extension on a semantic level, we *extend* the definition of the satisfaction $\models_{\mathcal{L}}$ (in $\mathcal{L}$) to a definition of satisfaction $\models_{\mathcal{F}}$ (in $\mathcal{F}$). This way, our definition is highly modular, language-independent and focuses on the description of dynamics (i.e. possible *state transitions*) as the central aspect that the functional description language $\mathcal{F}$ adds on top of state description language $\mathcal{L}$. It can be applied to various languages $\mathcal{L}$ in the very same way as it only requires a model-theoretic semantics for the static language $\mathcal{L}$ (which almost all commonly used logics provide). Furthermore, our model-theoretic approach coincides to the common understanding of functional descriptions to be *declarative* descriptions *what* is provided rather than *how* the functionality is achieved.

This deliverable is structured as follows: Chapter 2 will identify the usage scenarios and define the requirements on functional specifications of Web Services. Chapter 3 gives a brief overview of existing approaches, finally we define a semantic for capabilities that is intended to be used in WSML in Chapter 4.

2 Web Services and their Capability: an Overview

The aim of this section is to define the problem space that this deliverable addresses. We enumerate the different views that can be taken when looking at a functional description of a Web Service. Starting from a rather naive signature like description to a rich functional description including temporal invariants, each of them having its justification by enabling the modeler to express specific aspect of a service. Finally, we discuss the different use cases that we envision to use formal functional descriptions and conclude the section with non-functional requirements on capability descriptions.

2.1 Views on Capability

In this section we enumerate different views (or level of abstraction) on Web Service capabilities. By illustrating this we show that different formalizations of capabilities are possible depending on which level of abstraction a capability description is chosen.

2.1.1 Signature View

The simplest way of looking at the functionality of a service is a **signature like view** by just examining its inputs and outputs. Although normally only concerned with the explicit inputs and outputs in the form of some message exchange this view can be easily extended to also include specific concepts being present in the ontological description of the state of the world before and after execution of the service. However this view does not enable to model the functional relation between the inputs and outputs, but marly their types.

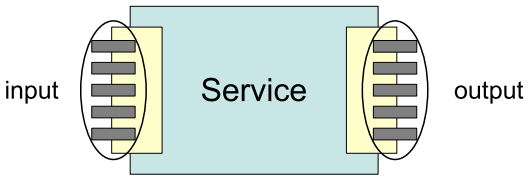

Figure 2.1: Signature View on Service Functionality

Example: Assuming a service that books airline tickets, a signature based view allows one to state that a flight service takes two locations (represented by strings), the desired arrival time and the names of the persons that want to travel as input. And as output a ticket is received. Also effects in the state of the world can be taken into account: e.g. an ontological representation of a seat reservation can be instantiated. However it is not possible to state relations between inputs and outputs, such that e.g. same Person supplied as input is also the one which appears in the reservation concept.

Application: Describing a service in this way is particular popular in the context of OWL-S service descriptions. Numerous works (e.g. [Paolucci et al., 2002]) have shown that when describing input an outputs using concepts of an ontology the inferences supported by the underlying ontology language (like subsumption reasoning) can be exploited e.g. in order to improve service discovery. It has been argued [Sycara et al., 1998] that other approaches to the description of capabilities, like in Software Specification expose (a) too much details and (b) the languages used are too complex to allow comparison of specification. One advantage of not describing the relation between inputs and outputs is that it is not not necessary to consider concrete input values that might be associated with a request for indexing a service repository. However the precision of this approach is limited.

2.1.2 Simple Functional View

A more fine grained view on the capability is taken when explicitly modeling the **functional relation between inputs and outputs**. In addition to typing inputs and outputs at this level of abstraction also the transformation function that represents the state transition can be described. This allows to infer which post state will hold depending on the pre state before service execution.

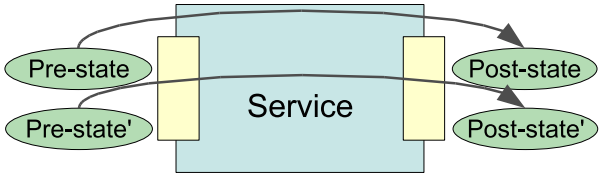

Figure 2.2: Functional View Relating Pre and Pos States

Example: Assuming the same airline ticket service mentioned before, a functional view allows besides typing information also to relate concrete pre states with resulting post states. By explicitly modeling the dependency it is possible to model that the same passenger specified in the booking request will also be the one for which the reservation holds.

Application: This level of abstractions allows one to model more details of the service and thus a higher degree of automation is possible. In [Kifer et al., 2004] we used logic programming rules for this task, however we have to acknowledge that it does not scale like the previous presented approaches, since a repository can not easily index all available services, but must consider always the concrete request for each service in order to perform discovery.

2.1.3 Functional View Including State Invariants

A further addition to the functional description can be made by stating state invariants, i.e. by stating which axioms are guaranteed to be true throughout the service execution.

Example: Taking the previous example this an invariant of interest could be for example to state that the personal data provided to perform a booking will

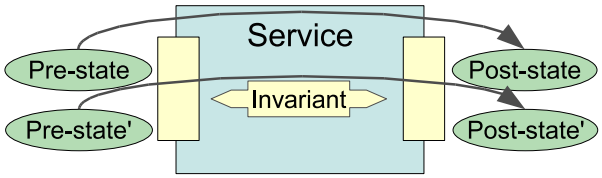

Figure 2.3: Functional View Including State Invariants

not be transferred to any other service or entity, but only be used by the ticket issuing company for the purpose of performing the travel arrangement.

Application: In principle sate invariants can be added to both previously presented approaches. They allow to make further statements about the internal state of the Web Service during execution, they may be particular useful during composition of components.

2.1.4 Functional View Including State Transition Invariants

Finally the most fine grained view on the capability one could take is to additionally specify constraints not only on the legal states during service execution, but also constraints on the sequence of legal states, e.g. to specify that a particular state can only occur after previous state has been reached.

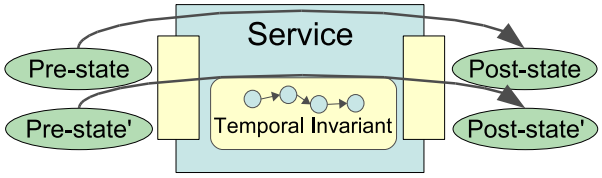

Figure 2.4: Functional View Include State Transition Invariants

Example: Taking the example of a hotel booking service, it can be of interest to describe constraints on the sequence of states. E.g to guarantee that the customers credit card is only charged, after he actually stays in the hotel in opposite to a service where the credit card is charged with a deposit immediately during the process of booking.

Application: This view requires an assertion languages that either includes already a mechanism for describing constraints on sequences of states or some methodology to encode it to a "static" knowledge representation language.

2.2 Usage Scenarios for Functional Descriptions

Here we enumerate those scenarios that might make use of the functional description provided by a WSMO capability.

2.2.1 Discovery

During Discovery the functional descriptions can enhance retrieval mechanism. Here we do not require that the specifications are complete, a human user (or another subsequent automated process) make the final selection and contracting of the actual service to be used.

2.2.2 Verification / Contracting

In this scenario we assume complete automation of the discovery process, or to use the functional specification as a means to verify an actual service implementation follows its advertisement. The information contained in the description must be sufficient to fully automate the discovery and subsequent invocation process.

2.2.3 Composition

Within this scenario we envision that a service request can not be fulfilled by any available service, therefore a composition of services must be performed. During composition the component (planner) must be able to retrieve "atomic" pieces of functionality and validate if they fit into the current plan...

2.3 Non Functional Requirements

After having enumerated the functional usage scenarios and criteria, this section summarizes the non-functional criteria we have to consider in our setting.

The final semantics presented in this document must be mapped into the WSML family of languages [Kifer et al., 2004].

The formal semantics should be intuitive, such that the resulting descriptions can be used also by domain experts.

The semantics should allow for the appropriate level of abstraction, that will depend on particular use case scenarios, i.e. ideally it should allow to describe a service at each of the abstraction levels outline before.

The semantics should allow to use existing reasoning infrastructure to solve the usage scenarios such as discovery and verification.

3 Existing Frameworks and Languages for Functional Specifications

In order to develop the semantics and syntax for the functional specification Web Services in this chapter we review existing work in the area. We split this analysis in two steps: First we survey formal specification languages (Section 3.1) and second the available frameworks (Section 3.2) that adopt a particular formalism and extend it with some methodology and tool support.

Functional specification (or capability specifications) of web services in WSMO follow the precondition/postcondition description style that has been invented by Hoare and is the basis of Hoare-style software verification calculi. Therefore we only survey frameworks that fit to this modeling style. If the semantics presented in this preliminary work do not find consensus in the working group we will extend this survey with e.g. Temporal Logic, Dynamic Logic, or Process Logic with respect to the formal languages and VDM, Z, Larch/LSL, SDL, and RAISE/RSL with respect to methodologies/frameworks.

3.1 Formal Languages

3.1.1 Hoare Calculi

Hoare logic [Hoare, 1969] is a formal system to provide a set of logical rules in order to reason about the correctness of computer programs with the rigour of mathematical logic. The central feature of Hoare logic is the Hoare triple. A triple describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form $\{P\}\ C\ \{Q\}$. where P and Q are assertions (usually expressed using predicate logic) and C is a command. P is called the precondition and Q the postcondition. The intuitive reading of such a triple is: Whenever P holds of the state before the execution of C, then Q will hold afterwards. Note that if C does not terminate, then there is no "after", so Q can be any statement at all. Indeed, one can choose Q to be false to express that C does not terminate. This is called "partial correctness". If C terminates and at termination Q is true, the expression exhibits "total correctness".

Hoare logic has axioms and inference rules for all the constructs of a simple imperative programming language. In addition to the rules for the simple language in Hoare's original paper [Hoare, 1969], rules for other language constructs have been developed since then by Hoare and many other researchers. This allows to move from expressions written in a programming language to logical expression, which are much more amendable to formal proofs.

To illustrate the axiomatization process of a program we will give two examples: First we will illustrate the case of a variable assignment in some programming language. Below axiom states that after the assignment any predicate holds for the variable that was previously true for the right-hand side of the assignment:

$$\overline{\{Q[V \to E]\}\ V := E\ \{Q\}}$$

The assignment rule has no condition. The rule states that if we want to show that Q holds after the assignment of V, then we must show that $Q[V \to E]$

holds before the assignment. I.e. the rule has to be red from the right to the left. If we consider the example: $\{y = 2\}x := 2\{y = x\}$. The postcondition is $y = x$. We do a substitution to get the precondition: $y = x[x \rightarrow 2]$. Which proves that the precondition must be $y = 2$.

The second example axiomatizes program sequences: The inference rules states that if you know that C_1 takes you from states satisfying P to states satisfying R, and that C_2 takes you from states satisfying R to states satisfying Q,then you know that executing C_1 and C_2 in sequence will take you from states satisfying P to states satisfying R.

$$\frac{\{P\}\ C_1\ \{S\} \wedge \{S\}\ C_2\ \{Q\}}{\{P\}\ C_1 \wedge C_2\ \{S\}}$$

Summary: The Hoare Calculi was the first formalism that used the pre and post condition style for describing software components. The main focus of the Hoare Logic is on axiomatizing various programming language constructs for verification. However despite 40 years of research this method received only little adoption in practice and only limited tools support is available. One reason for that might also be that even the axiomatization of simple programs turns out to result in very complex formulas.

3.1.2 Situation Calculus

The situation calculus is a methodology for specifying the effects of elementary actions in first-order classical logic. It was introduced by McCarthy [McCarthy, 1963]. The focus of situation calculus is to specify elementary actions. It is designed to reason about actions in the real world. The following explanation and example is taken form [Bonner and Kifer, 1998c].

We illustrate situation calculus taking the "Tower of Hanoi" example, in which a robot arm manipulates toy blocks sitting on a table top. A typical problem is to specify the effects of lifting a block, or of putting one block down on top of another. Such actions may have several different effects. For instance, when a robot picks up a block, the block changes position, the block beneath it becomes clear, and the robot hand becomes full. Actions may also have preconditions on their execution. For instance, a (one-armed) robot cannot pick up a block if its hand is already holding something. Specifying all the pre-conditions and all the effects of such actions is a central aim of the situation calculus. In the situation calculus, both states and actions are denoted by function terms. For instance, the function term $pickup(b)$ might denote the action of a robot picking up block b.

A function term denoting a database state is called a situation. The constant symbol s_0 denotes the initial state, before any actions have taken place. If s is a situation and a is an action, then the function term $do(a, s)$ is a situation denoting the state derived from s by applying action a. Thus, applying the actions a_1, a_2, a_3 to the initial state in that order gives rise to a state denoted by the situation $do(a_3, do(a_2, do(a_1, s_0)))$. Predicates whose truth depends on the situation are called fluents. For instance, the atomic formula $on(b_1, b_2, s)$ is a fluent. Intuitively, it means that block b_1 is on top of block b_2 in situation s. Moreover, for each action, we must say not only what it changes, but also what it does not change. The need to do this is illustrated in the following example:

```
//clear(x, s) in state s, there are no blocks on top of block x.
//on(x, y, s) in state s, block x is on top of block y.
//move(x, y) move block x on top of block y, provided that x and y are clear.

possible(move(X, Y ), S) ← clear(X, S) ∧ clear(Y,S) ∧ X ≠ Y
on(X, Y, do(move(X, Y ), S)) ← possible(move(X, Y ), S)
clear(Z, do(move(X, Y ), S)) ← possible(move(X, Y ), S) ∧ on(X, Z, S)
```

The first rule describes the pre-conditions of the move action. It says that it is possible to execute $move(X, Y)$ in situation S if blocks X and Y are clear. The next two rules describe the effects of the action, assuming the action is possible. The second rule says that after executing $move(X, Y)$, block X is on top of block Y. The third rule says that if block X is on top of block Z in situation S, then after executing $move(X, Y)$, block Z is clear. The above rules are called effect axioms, since they describe which facts are affected by the actions.

These rules are not sufficient, however, because in some situations, many other formulas might be true, but they are not logically implied by the effect axioms. To see this, consider a state described by the following atomic formulas:

$$clear(a, s_0) \wedge clear(b, s_0) \wedge clear(c, s_0) \wedge on(a, d, s_0) \wedge on(b, e, s_0)$$

The situation s_0 in these formulas indicates that they refer to the initial database state. Observe that block c is clear in this situation. Now, suppose we "execute" the action $move(a, b)$, thus changing the situation from s_0 to $do(move(a, b), s_0)$. Is block c still clear in the new situation? Our common sense says yes, since c should not be affected by moving a onto b. However, the formula $clear(c, do(move(a, b), s_0))$ is not a logical consequence of the set of formulas specified so far. In order to make $clear(c, do(move(a, b), s_0))$ a logical consequence of the specification, we need formulas that say what fluents are not changed by the action $move(a, b)$. These formulas are called frame axioms. In general, a great many things are not changed by an action, so there will be a great many frame axioms. Specifying all the invariants of an action in a succinct way is known as the frame problem. Early solutions to this problem required one axiom for each action-fluent pair [C.Green, 1969]. In general, specifying frame axioms in this way requires $O(M + N)$ rules, where N is the number of actions, and M is the number of fluents.

Summary Situation calculus is not a language in it self but rather an encoding of dynamics in classical first order logic. It is based on elementary actions and states. Lots of research has been done around the so called "frame problem", which essentially is about finding a concise amortization of those facts that do not change during a state transition.

3.1.3 Transaction Logic

Transaction Logic (abbreviated $\mathcal{TR}$) is a formalism designed to deal with the phenomenon of state changes in logic programming, databases, and AI. Transaction Logic has a model theory and a sound and complete proof theory. Unlike many other logics, however, it is suitable for programming procedures that accomplish state transitions in a logically sound manner. Transaction logic amalgamates such features as hypothetical and committed updates, dynamic constraints on transaction execution, nondeterminism, and bulk updates.

$\mathcal{TR}$ has a "Horn" version that has both a procedural and a declarative semantics. It was designed with several application in mind, especially in databases, logic programming, and AI. It was therefore developed as a general

logic, so that it could solve a wide range of update related problems, this includes especially tests and conditions on actions, including preconditions, postconditions, and intermediate conditions. For a wide class of problems, $\mathcal{TR}$ avoids the frame problem. The frame problem arises because, to reason about updates, one must specify what does not change, as well as what does.

Moreover $\mathcal{TR}$ is also designed to describe nondeterministic actions. For instance, the user of a robot simulator might instruct the robot to build a stack of three blocks, but he may not tell (or care) which three blocks to use, but some constraint like that within one stack a lower block must be bigger then the ones on top of it. $\mathcal{TR}$ enables users to specify what choices are allowed and can leave particular implementation details away.

All actions (except elementary actions) are characterized not only by its initial and final state but also by a sequence of states inbetween (the so called *executionpath*). This allows to express constraints over state sequences.

Syntactically, $\mathcal{TR}$ extends First-order classical logic with a new logical operator called serial conjunction, denoted $\otimes$. Intuitively, if α and β are $\mathcal{TR}$ formulas representing programs, then the formula $\alpha \otimes \beta$ also represents a program, one that first executes α and then executes β. $\mathcal{TR}$ also interprets the connectives of classical logic in terms of action. For instance, the formula $\alpha \vee \beta$ intuitively means "do α or do β" and the formula $\neg\alpha$ intuitively means "do something other than α." In this way, $\mathcal{TR}$ uses logical connectives to build large programs out of smaller ones.

Semantically, formulas in $\mathcal{TR}$ are evaluated on paths, i.e., on sequences of states. Intuitively, if ϕ is a $\mathcal{TR}$ formula representing an action, then ϕ is true on a path s_1, s_2, ..., s_n if the action can execute on the path, i.e., if it can change a state from s_1 to s_2 to ... to s_n.

When used for reasoning, properties of programs are expressed as $\mathcal{TR}$ formulas, and a logical inference system derives properties of complex programs from those of simple programs [Bonner and Kifer, 1998b]. In this way, one can reason, for instance, about whether a transaction program preserves the integrity constraints of a database. One can also reason about when a program will produce a given outcome, when it will abort. In addition, one can reason about actions in an AI context. For instance, using $\mathcal{TR}$, a user can axiomatize the elementary actions of a robot and reason about them. In AI terminology, this reasoning takes place in open worlds, that is, in the absence of the closed world assumption. The assumption of open worlds separates the $\mathcal{TR}$ theory of reasoning [Bonner and Kifer, 1998b] from the $\mathcal{TR}$ theory of logic programming [Bonner and Kifer, 1998a], which is based on closed worlds.

In $\mathcal{TR}$, all transactions are a combination of queries and updates. Queries do not change the state of the world, and can be expressed in classical logic. In contrast, updates do changes to the state of the world and are expressed in an extension of classical logic. The simplest kind of updates are called elementary state transitions. In principle, there are no restrictions on the changes that an elementary update can make, however usually insertion and deletion of atomic formulas are used as canonical examples of elementary updates. Usually the updates are represented by a special set of predicate symbols (*.ins* and *.del*).

```
move(X,Y) ← pickup(X) ⊗ putdown(X,Y)
pickup(X) ← claer(X) ⊗ on(X,Y) ⊗ on.del(X,Y) ⊗ clear.ins(Y)
putdown(X,Y) ← wider(Y,X) ⊗ clear(Y) ⊗ on.ins(X,Y) ⊗ clear.del(Y)
```

Summary The semantics of $\mathcal{TR}$ is based on paths rather then on pairs of states, in exact it is very good at describing and querying different sequences of states. $\mathcal{TR}$ uses oracles to represent atomic actions such as delete and insert

facts and therefore avoids the frame problem. Most of the research around $\mathcal{TR}$ has been carried out in the segment of $\mathcal{TR}$ falling into Logic Programming.

3.2 Existing Frameworks

There are several specification frameworks which are used in academia and the industry, for instance Eiffel, UML/OCL , VDM, B, Z, Larch/LSL, SDL, RAISE/RSL. In the following we will have a look at some of them and explain how they deal with functional specifications of software components and particularly how input values are represented in these specification.

Since in WSMO functional specifications are based on pre- and postconditions, we restrict ourselves in the following to specification frameworks that are based on pre- and postconditions. This excludes for instance the well-known formalisms of *Abstract State Machines* (ASMs) as well as the *B Method*.

3.2.1 Eiffel

Eiffel [Meyer, 1992; Eiffel-Software, 2004] has been invented by Bertrand Meyer as a systematic approach to object-oriented software construction [Meyer, 2000]. It can be considered as a programming language with specific language constructs for the functional specification of methods (that is pre- and postconditions) as well as class invariants. Eiffel embodies the so-called *Design by Contract* principle, where the signature of a interface as well as the functional specification of the interface are considered as a contract between the client using the interface and the provider of the concrete implementation of the interface. None of the parties is allowed to break the specified contract, or more precisely contracts can be checked in a concrete system during runtime and broken contracts are detected and reported. Together with a suitable tool environment and a process recommendation, Eiffel can be considered not only as a programming language but as a formal methodology for software development.

The notion of a contract. In human affairs, contracts are written between two parties when one of them (the supplier) performs some task for the other (the client). Each party expects some benefits from the contract, and accepts some obligations in return. Usually, what one of the parties sees as an obligation is a benefit for the other. The aim of the contract document is to spell out these benefits and obligations.

A tabular form such as the following (illustrating a contract between an airline and a customer) is often convenient for expressing the terms of such a contract:

	Obligations	Benefits
Client	(Must ensure precondition): Be at the Santa Barbara airport at least 5 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.	(May benefit from postcondition): Reach Chicago.
Supplier	(Must ensure postcondition): Bring customer to Chicago.	(May assume precondition) No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price.

A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.

The same ideas apply to software. Consider a software element E. To achieve its purpose (fulfil its own contract), E uses a certain strategy, which involves a number of subtasks, $t_1, \ldots, t_n$. If subtask t_i is non-trivial, it will be achieved by calling a certain routine R. In other words, E contracts out the subtask to R. Such a situation should be governed by a well-defined roster of obligations and benefits – a contract.

Assume for example that t_i is the task of inserting a certain element into a dictionary (a table where each element is identified by a certain character string used as key) of bounded capacity. The contract will be:

	Obligations	Benefits
Client	(Must ensure precondition): Make sure table is not full and key is a non-empty string.	(May benefit from postcondition): Get updated table where the given element now appears, associated with the given key.
Supplier	(Must ensure postcondition): Record given element in table, associated with given key.	(May assume precondition) No need to do anything if table is full, or key is empty string.

This contract governs the relations between the routine and any potential caller. It contains the most important information that can be given about the routine: what each party in the contract must guarantee for a correct call, and what each party is entitled to in return.

An example of a interface specification. In order to benefit and support the concept of contracts, contracts need to be formalized and explicitly documented. In the spirit of seamlessness (encouraging us to include every relevant information, at all levels, in a single software text), in Eiffel the routine text is equipped with a listing of the appropriate conditions.

Assuming the routine is called `put`, it will look as follows in Eiffel syntax, as part of a generic class `DICTIONARY [ELEMENT]`:

```
class DICTIONARY
[ELEMENT]
  put (x: ELEMENT, key: STRING) is
   require
      count <= capacity
      not key.empty
   do
     [... Some insertion algorithm ...]
   ensure
      has (x)                  -- has(x) returns true afterwards
      item (key) = x           -- x can be retrieved with the key using the item method
      count = old count + 1    -- the dictionary has one more entry afterwards
   end
   [...]
  get (key: STRING):ELEMENT is
   require
      not key.empty
      has\_key(key)
   do
     [... Some retrieval algorithm ...]
    Result := ...
   ensure
      Result = item (key)
      count = old count
   end
end
```

Listing 3.1: An Example Eiffel Specification

Let's have a look at the `put` specification: The require clause introduces an input condition, or precondition, the ensure clause introduces an output condition, or postcondition. Both of these conditions are examples of assertions, or logical conditions (contract clauses) associated with software elements. In the precondition, count is the current number of elements and capacity is the

maximum number, in the postcondition, has is the boolean query which tells whether a certain element is present, and item returns the element associated with a certain key. The notation old count refers to the value of count on entry to the routine.

In the `get` contract we have another distinct keyword, namely `Result`. This keyword refers to the object or value which is returned at the end of a successful execution of this method. The keyword is the same as the one which is used in the programming language part in the `do` clause.

The assertion language in Eiffel has been designed in such a way that the (boolean) conditions can be effectively evaluated during runtime. That means that predicates usually refer to boolean methods. The assertion language seems to be less expressive than assertion languages in other specification frameworks that are not directly based on a programming language and the idea of runtime checking of contracts.

From this prototypical example, we can observe the following:

- Input variables are explicitly declared in the contract (more precisely in the signature part).
- There is an explicit representation of the output value (`Result`) which allows to refer to the return value in the postcondition
- By using the keyword `old`, one can refer to the value of some (boolean) property before the execution of a method. This keyword can only be used in post conditions.

3.2.2 The Unified Modeling Language and OCL

The Unified Modelling Language [**?**; Object Management Group, 2005] is a graphical language for the modelling and description of systems, in particular software systems. It is a collection of around eight different diagram types which capture different perspectives on the system under consideration like the logical system structure (Class Diagrams), typical usage scenarios (Use Case Diagrams), interactions between the elements the system constitutes of (Interaction Diagrams), the life-cycle of the single elements (State Diagrams), the physical structure of the system (Deployment Diagrams) and so forth. The UML evolved and unified different preexisting and competing proposals for object-oriented modelling languages for software systems and became the standard object-oriented modelling language in the software engineering domain maintained by the Object Management Group (OMG).

The Object Constraint Language (OCL) is an integral part of the UML Standard since UML version 1.1. UML has been equipped with OCL in order to allow modelers to express unambiguously nuances of meaning that the graphical elements of UML can not convey by themselves. An important (and even the first) application for OCL was the Specification of the UML Metamodel itself which was recognized as being ambiguous in the natural language based specification of UML version 1.0. It is a formal language for the specification of functional behavior of the single elements of a software system as well as global constraints on valid system states. OCL supports preconditions, postconditions for functional specifications of methods and class invariants for the description of global constraints on the system state.

It is important to notice that UML/OCL models are not bound to any specific implementation language. Indeed, they are implementation language independent.

Context of OCL Constraints. OCL expression have no meaning by themselves. In general, they can only be used to constrain the standard semantics of modelling elements of the UML and to detail out the precise meaning of these standard elements in a specific model (if such an derivation from the standard semantics of UML and its elements is needed). Thus, OCL Expression always refer to specific elements in a given UML Model. Such a UML model usually is a class diagram. Other diagram types such as state diagrams are only weakly supported by OCL at present. When a modeler describes a constraint, he must always specify to which element (for example which class) the constraint refers. This element is called the *context* element of a constraint. For some constraints, the specific choice of the context is not relevant from a semantic perspective. Some of the build in stereotypes can be considered as graphical shortcuts of (stereotypical) OCL Constraints, for instance the `<<complete>>` or `<<distinct>>` stereotypes of inheritance relationships.

OCL Constraints OCL is a typed language which can be consider as some sort of ,,object-oriented predicate logic" whereby quantors can only quantify over finite domains[1].

The readability of the language for the average software engineer had been emphasized during the development of the language. For instance, path-expression which are common in object-oriented programming and specification languages are supported by OCL as well.

OCL supports as constraints mainly invariants on class diagramms as well as fuctional specifications of methods by preconditions and postconditions. Roughly spoken, all these constraints are basically a boolean OCL expression (the assertion language) combined with a context declaration. In the case of method specifications, the context declaration includes the method signature.

The general template of a functional specification of a methods looks as follows (whereby the method takes parameters $p_1, \ldots, p_N$ with the respective types $T_1, \ldots, T_N$ as input parameters and (possibly) returns a value of type $ResT$)[2]:

```
context (cxtName)?
  className::methodName(p1:T1,...,pN:TN)(: ResT)?
     (pre  (constraintName)? : BooleanOCLExpression)?
     (post (constraintName)? : BooleanOCLExpression)?
```

In preconditions, the parameters can be used $p_1, \ldots, p_N$ like (OCL-)variables. The semantics of these variables is identical to universally quantified variables. The quantors of these variables are not specified explicitly but determined implicitly by the declaration of a method signature. The declaration allows tools that deal with the OCL constraints to distinguish between input values and other properties from class diagrams (like attributes of classes or relations between classes)

In postconditions, the input parameters $p_1, \ldots, p_N$ can be used in the very same way. Moreover there are two additional syntactic constructs which can be used in postconditions only: (i) In order to be able to describe the result value which is returned by a method one can use the literal (or keyword) `result` and (ii) In order to be able to refer to values of properties of the system in the prestate of the method call for each property `prop` (for instance, predicates and attributes, but not OCL variables) one can use the property `prop@pre`.

[1] In fact, it has been shown by Peter Schmitt [Schmitt, 2001] that the logic underlying OCL is strictly more expressive over finite domains than First-order predicate logic

[2] Optional parts of the template are denoted by (...)?

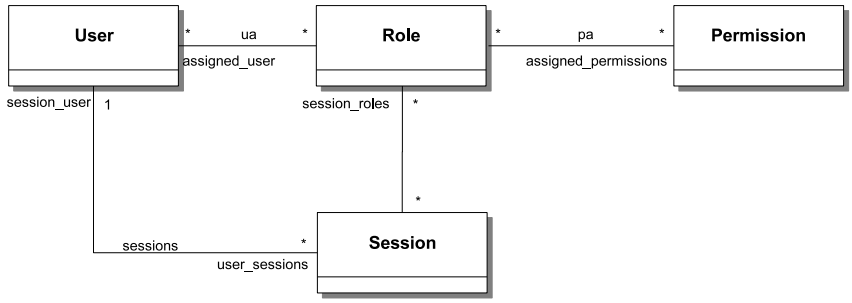

Figure 3.1: A simple UML diagram for a Role-based Access Control Scenario.

An example of a interface specification. Lets consider a Role-based Access Control Scenario illustrated in Figure 3.1: A user works and can take actions in the context of a user session. A user can participate in multiple sessions during each point in time. Each user has a set of roles assigned, these are the roles the user principally can realize. Every session has a single user assigned and a set of roles which the respective user of the session currently is allowed to realize. Each role provides a set of assigned permissions as well as a set of persons which are registered for and allowed to realized this role.

This model can be used as a starting point to describe a system which supports the administration and implementation of access-control policies based on roles.

Now, lets assume that class `user` has a method `assignUser(r:Role)` which can be used to assign a new role to a instances of class `user`. The method registers the user for this role only if the user does not already have the role and returns the boolean value `true` in this case.

A formal specification of the functionality of this method in OCL could be as follows:

```
context u:User::assignUser(r:Role):boolean

--- the user u (for which the method has been called) does not already have the given role assigned.
pre: not(r.assigned_users->includes(u))

--- After the execution the set of assigned users is precisely the one we had before the invocation
--- including the user u (for which the method has been called) and return the boolean value 'true'.
post: r.assigned_users = r.assigned_users@pre->including(u)
  and result = true
```

Listing 3.2: Example OCL Specification

OCL supports in particular collection types (like the set `r.assigned_users` of assigned users to a role `r`) and corresponding standard operations (like union, intersection, contains, etc). In particular it is possible to iterate through such collections and quantify over such collection.

For instance, if we would have to specify a method which adds a creates user instance for a given name to the system in case that a user with the same name does not already exist, then we could use the following OCL specification:

```
context User::addUser(c:Name)

--- There is no instance u1 of class User with the given name c
pre: User.allInstances->forAll(u1| u1.name <> c)
```

```
-- There an instance u1 of class User with the given name c which has not been there before the
-- method call and the extension of class User has only changed by this specific instance u1
post: User.allInstances ->exists(u1|
  u1.name = c and
  User.allInstances@pre ->excludes(u1) and
  User.allInstance  = User.allInstance@pre ->including(u1))
```

Listing 3.3: Example OCL Specification

In contrast to Eiffel, the assertion language if OCL has a more abstract flavor and is less oriented towards concrete implementations of the model. In particular associations can be realized in general in various ways in a concrete implementation of the same UML model. OCL abstracts from these differences and treat associations (and some other modelling elements of UML) as explicit modelling elements which do not have to have directly correspondent elements in a programming language.

3.3 Conclusion

We have briefly surveyed some of the most-popular and widely-used frameworks for functional specifications of software components in the industry as well as academia.

Besides the differences in the syntactic details and the background of the single specification frameworks, we can extract basically the following general pattern which is independent from the specific assertion language that is used in the specification frameworks:

- There are syntactic constructs to refer to prestate-values, e.g. `old property`, `property@pre`. These constructs can only be used in post conditions.
- Some syntactic constructs to refer to the output, e.g. `Result, result`
- Input values are declared explicitly and are usually interpreted variables. On a semantic level these variables is the whole capability description, that means using the same variable names in pre- and postconditions means refering to the same (input) values. variables are universally quantified. Their quantors are not explicitly declared by determined on a meta-level by the semantics of a functional specification based on pre- and postconditions. The scope of these variables
- A component specification is a logically indivisible unit. The functionality description depends on both, pre- and postconditions, the single logical expressions which describe these conditions themselves are meaningless (outside the functional specification as a coherent unit)

4 Semantics of Functional Specification

In the previous chapter we have surveyed several well-known approaches for the functional specification of software components. In this chapter we propose to apply the patterns found to the area of functional web service description in WSML.

Additionally, some extensions are proposed, that are relevant to some of the intended applications and to go beyond what can be expressed by means of preconditions and postconditions. More specifically, we include syntactic means to express guarantees that refer to an execution of the described Web Service (i.e. the whole sequences of states that are passed when executing the Web Service), that is *invariants* that are *local* to Web Service executions. We propose a new keyword `throughout` to be used in capability descriptions for this purpose.

Finally, we define a formal semantics for such capabilities in WSML. We use a model-theoretic approach for this purpose. More precisely, we show how to extend an arbitrary assertion language for expressing properties of and requirements on single *states* of the world to a language that can be used to describe the capability of a Web Service in terms of possible state transitions (or executions, resp.). Thus, we extend a language for describing a static world to a language which can deal with dynamic aspects of a world. To achieve this, the purely syntactical extension of the assertion language to describe static aspects (states) to a capability description language to capture dynamics (execution paths) is accompanied by an extension of the model-theoretic semantics of the assertion language to a model-theoretic semantics for our capability description language.

4.1 Towards A Formal Model of Web Services

This section presents a model which allows for a precise definition of the notion *Web Service*. This model will later be used in Section 4.2 to define the formal semantics of a capability description of Web Service in WSMO. The model is not based on any specific logical formalism and thus can be formally represented in various logics of sufficient expressivity to enable reasoning with semantic descriptions of Web Service capabilities.

In future versions of this document, we will instantiate this basic model with the existing semantic framework of the WSML family of languages and identify a layering that is suitable for the identified requirements.

First we will illustrate our assumptions on the domain (Web Service descriptions) in an intuitive fashion, such that the formal definitions in Section 4.2 are easier to read.

4.1.1 A changing world

We consider the world as an entity that changes over time. Entities that act in the world - which can be anything anything from a human user to some computer program - can affect how the world is perceived by themselves or other

entities at some specific moment in time. At each point in time, the world is in one particular state that determines how the world is perceived by the entities acting therein. We need to consider some language for describing the properties of the world in a state. In the following we assume an *arbitrary* (but fixed) signature Σ that usually based on domain ontologies, and some language $\mathcal{L}(\Sigma)$.

We use classical First-order Logic for illustration purpose which can easily be applied to other languages such as WSML or OWL. Consider a signature $\Sigma \supseteq \{isAccount(\cdot), balance(\cdot), \geq, 0, 1, 2, \ldots\}$ that denotes bank accounts and their balance. It allows comparing the respective values $\mathcal{L}(\Sigma)$, for instance containing expressions like $\forall ?x.(isAccount(?x) \Rightarrow balance(?x) \geq 0)$ stating that the balance of any account needs to be non-negative. In the context of dynamics and properties of the world that can change, it is useful to distinguish between symbols in Σ that are supposed to have always the same, fixed meaning (e.g. $\geq$, 0) and thus can not be affected by any entity that acts in the world, and symbols that can be affected and thus can change their meaning during the execution a Web Service (e.g. $isAccount(\cdot)$, $balance(\cdot)$). We refer to the former class of symbols by *static symbols* (denoted by Σ_S) and the latter by *dynamic symbols* (denoted by Σ_D).

4.1.2 Abstract State Spaces

We consider an abstract state space $\mathcal{S}$ to represent all possible states s of the world. Each state $s \in \mathcal{S}$ completely determines how the world is perceived by each entity acting in $\mathcal{S}$. Each statement $\phi \in \mathcal{L}(\Sigma)$ of an entity about the (current state of) the world is either true or it is false. [1]. Thus, a state $s \in \mathcal{S}$ in fact *defines* an interpretation $\mathcal{I}$ (of some signature Σ). However, not all Σ-Interpretations $\mathcal{I}$ represent senseful observations since $\mathcal{I}$ might not respect some ,,laws" that the world $\mathcal{S}$ underlies, e.g. that the balance of any bank account is not allowed to be negative. In the following, we assume that these laws are captured by a background ontology $\Omega \subseteq \mathcal{L}(\Sigma)$ and denote the set of Σ-Interpretations that respect Ω by $Mod_\Sigma(\Omega)$. Considering our example signature from above, the following interpretations denote states $s \in S$:

$$s_0 : balance(acc_1) = 10 \wedge balance(acc_2) = 100$$
$$s_n : balance(acc_1) = 30 \wedge balance(acc_2) = 80$$

4.1.3 Changing the World

By means of well-defined change operations, entities can affect the world that denote state transitions in $\mathcal{S}$. In our setting, these change operations are single concrete *executions* of Web services W. Following [Keller and Lara (eds.), ; Keller et al., 2005], a change operation is represented by a *service* S that is accessed via a Web service W. S is achieved by executing W with some given input data $i_1, \ldots, i_n$ that specify *what* kind of particular *service* S accessible via W is requested by the client, i.e. $S \approx W(i_1, \ldots, i_n)$.

Given input data $i_1, \ldots, i_n$, the execution of a Web service W essentially causes a state transition τ in $\mathcal{S}$, transforming the current state of the world

[1] We consider classical logic (and thus only *true* and *false* as truth values) here. However, the presented model can be used as it is in the context of non-classical logics by just considering a different class of interpretations $\mathcal{I}$, e.g. in the case of multi-valued logics, we can use multi-valued interpretations.

$s \in \mathcal{S}$ into a new state $s' \in \mathcal{S}$. However, a transition τ will in general not be an *atomic transition* $\tau = (s, s') \in \mathcal{S} \times \mathcal{S}$ but a sequence $\tau = (s_0, \ldots, s_n) \in \mathcal{S}^+$, where $s_0 = s$, $s_n = s'$ and $n \geq 1$. In every intermediate state s_i in τ some effect can already be perceived by an enitity. This is especially relevant for Web services that allow accessing long lasting activities that involve multiple conversation steps between the requester and the Web service W. Please note, that $\tau = (s_0, \ldots, s_n) \in \mathcal{S}^+$ implies that we assume Web Service executions to *terminate*. We consider this assumption as a useful one, that should be met in all practical application scenarios.

Let us consider some international bank transfer having as concrete input data the information to transfer \$20 from acc_1 to acc_2. The model of the world might have between s_0 and s_n the following intermediate state:

$$s_1 : balance(acc_1) = 10 \wedge balance(acc_2) = 80$$

4.1.4 Outputs as Changes of an Information Space

During the execution $W(i_1, \ldots, i_n)$ of a Web service W, W can send some information as output to the requester. We consider these outputs as updates of the so-called *information space* of the requester of a service S. More precisely, we consider the *information space* of some service requester as a set $IS \subseteq U$ of objects from some universe U. Every object $o \in IS$ has been received by the requester from W during the execution $W(i_1, \ldots, i_n)$. During the execution the information space itself evolves: starting with the empty set when the Web service is invoked the execution leads to a monotonic sequence of information spaces $\emptyset = IS_0 \subseteq IS_1 \subseteq \ldots \subseteq IS_k$. Monotonicity of the sequence models that information that has been received by the user will not be forgotten until service execution completion.

Within our bank transfer example, during some transaction we might receive first a message acknowledgment, and then a confirmation that the transaction has been approved and initialized.

$$IS \models ack(20051202, msgid23)$$
$$confirm(acc_1, acc_2, 20)$$

4.1.5 Observations in Abstract States

Our aim is to describe *all* the effects of Web service executions for a requester. Obviously, a requester can observe in every state $s \in \mathcal{S}$ world-related properties represented by statements ϕ in $\mathcal{L}(\Sigma)$ that hold in s. Additionally, he can perceive the information space $IS \subseteq U$ described above. Thus, the abstract state space $\mathcal{S}$ in a sense ,,corresponds" to the observations that can be made in s, namely all pairs of Σ-interpretations $\mathcal{I} \in Mod_\Sigma(\Omega)$ and (possible) information spaces $IS \subseteq U$. Consequently, we represent the observations related to a state s by an observation function $\omega : \mathcal{S} \to Mod_\Sigma(\Omega) \times \mathcal{P}(U)$ that assigns every state $s \in \mathcal{S}$ a pair $(\mathcal{I}, IS)$ of a Σ-interpretation $\mathcal{I}$ (respecting the domain laws Ω) and an information space IS. We denote the first component of $\omega(s)$ by $\omega_{rw}(s)$ (*real-world properties*: how an entity perceives the world) and the second component by $\omega_{is}(s)$ (*information space*: how the invoker perceives the information space).

However, we require the observation function ω to be a (fixed) total function as it can *not* be arbitrary. This means that the observations $\omega(s)$ of any entity are well-defined in *every* abstract state s. Moreover, any perception representable in terms of $\mathcal{L}(\Sigma)$ and U that is consistent with the domain model Ω should actually corresponds to some abstract state $s \in \mathcal{S}$ by means of ω, so that ω is surjective. However, since we assume a fixed signature Σ and thus a limited language for describing observations about the world, we do not assume that ω is injective, i.e. there could be distinct states s, s' of the world which can *not* be distinguished by the (limited) language $\mathcal{L}(\Sigma)$, i.e. $\omega_{rw}(s) = \omega_{rw}(s')$.

The former means that a state $s_i : \mathit{belance}(\mathit{acc}123) = -10$ is no model since it is inconsistent with the domain ontology (we previously required the balance to be positive). The latter determines that there is always a corresponding abstract state to a set of sentences. However not all states in $\mathcal{A}$ can be distinguished. E.g. if we model the transfer of money between two accounts and do not include details of the transaction system, we can not express the states between the initialization of the transaction and its commit states. However there are intermediate states, and only by the limitations of $\mathcal{L}(\Sigma)$ we can not distinguish them.

4.1.6 Web Service Executions

Given some input $i_1, \ldots, i_n$, the Web service execution $W(i_1, \ldots, i_n) = (s_0, \ldots, s_m)$ starting in state s_0 induces a sequence of observations $(\omega(s_0), \ldots, \omega(s_m))$ which can be made by the service requester during the execution. However, not all such sequences τ of abstract states actually do represent a meaningful state-transition caused by an execution of W. For τ to faithfully represent some $W(i_1, ..., i_n)$ we need to require at least that for any two adjacent states s, s' in $W(i_1, \ldots, i_n)$ some change can be observed by the invoker, and that objects which are in the information space (i.e. have been received by the invoker) at some point in time during the execution can not disappear until the execution is completed. As discussed later, in general we need to require some further constraints on a sequence τ such that we can interpret τ as a possible run $W(i_1, \ldots, i_n)$ of a Web service W. We call s_0 the pre-state of the execution, s_m the post-state of the execution, and all other states in $W(i_1, ..., i_n)$ intermediate states.

$$
\begin{aligned}
s_0 &: \mathit{balance}(\mathit{acc}_1) = 10 \wedge \mathit{balance}(\mathit{acc}_2) = 100 \\
s_1 &: \mathit{balance}(\mathit{acc}_1) = 10 \wedge \mathit{balance}(\mathit{acc}_2) = 80 \\
s_n &: \mathit{balance}(\mathit{acc}_1) = 30 \wedge \mathit{balance}(\mathit{acc}_2) = 80
\end{aligned}
$$

In the case of the transaction example s_0 is the pre state with the initial balances. In the intermediate state s_1 the balance of acc_2 has already been reduced by \$20, but acc_1 has not yet been increased. s_n is the post-state of the transaction where the money transfer has succeeded.

4.1.7 Web Services

A Web service W then can be seen as a set of executions $W(i_1, \ldots, i_n)$ that can be delivered by the Web service in any given state of the world to a requester when being equipped with any kind of valid input data $i_1, \ldots, i_n$. However, in order to keep track of the input data that caused a specific execution, we need to represent a Web service in terms of a slightly richer structure than a

set, namely a mapping between the provided input values $i_1, \ldots, i_n$ and the resulting execution [2] $W(i_1, \ldots, i_n)$.

In our running example the Web service of an bank accepting two account numbers (i_1, i_2) and some amount (i_3) can yield as example. For all valid accounts (given a sufficient initial balance) it will transfer the amount (i_3) from i_1 to i_2. And thus the actual Web services corresponds to a set of state transitions (and not only one), where each transition is determined by the concrete input values supplied.

Figure 4.1 illustrates the presented model. The Web service W provides four different concrete services $(S_1 \ldots S_4)$. Each single state is determined by the two components of ω - the information space and the real world. The Web service is a set of possible transitions that is denoted by a dark green area inside the abstract state space.

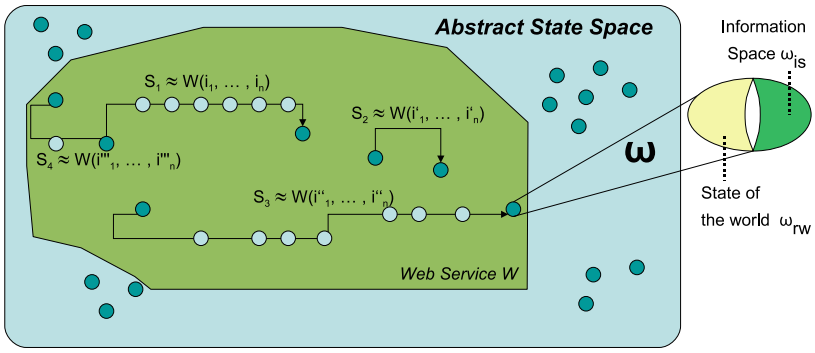

Figure 4.1: An abstract Model of the World and Web services therein.

4.1.8 Functional Description of Web Services

As a part of rich model description frameworks like OWL-S [Martin (ed.), 2004], SWSF [Martin et al., 2005] and WSMO [Lausen et al., 2005], functional descriptions of Web services $\mathcal{D}_\mathcal{W}$ are *syntactic expressions* in some specification language $\mathcal{F}$ that is constructed from some (non-logical) signature $\Sigma^\mathcal{F}$. Each expression $\mathcal{D} \in \mathcal{F}$ expresses specific requirements of Web services W, usable to constrain the set of all Web services to some subset that is interesting in a particular context. Hence, the set of Web services W that satisfy the functional description D (denoted by $W \models_\mathcal{F} \mathcal{D}$) can be considered as actual meaning of $\mathcal{D}$. This way, we can define a natural *model-theoretic semantics* for functional descriptions by defining a satisfaction relation $\models_\mathcal{F}$ between Web services and functional descriptions.

In general, various simpler syntactic elements are combined within a functional description $\mathcal{D} \in \mathcal{F}$. State-based frameworks as the ones mentioned above use at least preconditions and postconditions. Whereas $\mathcal{D}$ refers to Web services, these conditions refer to simpler semantic entities, namely *states*, and thus in a sense to a static world. Such state conditions ϕ are expressions in (static) language $\mathcal{L}$ over some signature $\Sigma^\mathcal{L}$. Single states s determine the truth value of these conditions. Formally, we have a satisfaction relation $\models_\mathcal{L}$ between states s and state expressions ϕ, where $s \models_\mathcal{L} \phi$ denotes that ϕ holds in state s. We observe that a language $\mathcal{L}$ that captures static aspects of the world is extended to a language $\mathcal{F}$ that captures dynamic aspects of the world on a syntactic level.

[2] This implies that we use a *deterministic* model for Web Services here. An extension to a non-deterministic perspective is straightforward.

In order to define a similar extension on a semantic level, we *extend* the definition of the satisfaction $\models_{\mathcal{L}}$ (in $\mathcal{L}$) to a definition of satisfaction $\models_{\mathcal{F}}$ (in $\mathcal{F}$). This way, our definition is highly modular, language-independent and focuses on the description of dynamics (i.e. possible *state transitions* τ) as the central aspect that the functional description language $\mathcal{F}$ adds on top of state description language $\mathcal{L}$. It can be applied to various languages $\mathcal{L}$ in the very same way as it only requires a model-theoretic semantics for the static language $\mathcal{L}$ (which almost all commonly used logics provide). Furthermore, our model-theoretic approach coincides to the common understanding of functional descriptions to be *declarative* descriptions *what* is provided rather than *how* the functionality is achieved.

Combining state-related descriptions with a functional description (or capability) essentially creates a constraint on possible Web Service executions. Executions of a Web service W whose capability has been described in terms of a capability description $\mathcal{D}$ (where $\mathcal{D}$ contains a prestate-constraint ϕ^{pre} and a post-state constraint ϕ^{post}) can no longer be arbitrary possible executions τ in an abstract state space $\mathcal{S}$, but whenever the prestate s_0 of τ respects ϕ^{pre} then the final state s_m of τ must respect ϕ^{post}. Otherwise, τ is not considered to represent an actual execution of a Web service W with capability $\mathcal{D}$.

4.2 Abstract State Spaces and Web Services

We will now give a series of definitions which capture the preceding semi-formal discussion in a rigorous way.

In the following, let Σ be some signature, $\mathcal{L}(\Sigma)$ be some logic over signature Σ and $\Omega \subseteq \mathcal{L}(\Sigma)$ be some background theory capturing relevant domain knowledge. Let $\Im(\Sigma)$ denote the set of Σ-interpretations in $\mathcal{L}(\Sigma)$ and $\Im(\Sigma,\mathcal{U})$ denote the set of Σ-interpretations such that for all $\mathcal{I} \in \Im(\Sigma)$ the universe considered in $\mathcal{I}$ (denoted by $universe(\mathcal{I})$) is a subset of $\mathcal{U}$. For a set U we use $\mathcal{P}(U)$ to denote the powerset of U. Let $Mod_{\Sigma}(\Omega,\mathcal{U})$ denote the Σ-Interpretations $\mathcal{I} \in \Im(\Sigma,\mathcal{U})$ which satisfy the domain model Ω (i.e. $\mathcal{I} \models_{\mathcal{L}(\Sigma)} \Omega$). We denote the meaning of a symbol $\alpha \in \Sigma$ that is assigned by an interpretation $\mathcal{I}$ by $meaning_{\mathcal{I}}(\alpha)$.

Given a signature $\Sigma_0 = \Sigma_D \cup \Sigma_S$ that is partitioned into a set Σ_D of dynamic symbols and a set Σ_S of static symbols, we extend Σ_0 to a signature Σ by adding a (new) symbol α_{pre} for each $\alpha \in \Sigma_D$. The set of these pre-variants of symbols is denoted by Σ_D^{pre}. Furthermore, add a new symbol *out* to Σ_0. The intention is as follows: Σ_S contains symbols that are interpreted always in the same way (static symbols), Σ_D contains symbols whose interpretation can change during the execution of a Web service (dynamic symbols), and Σ_D^{pre} contains symbols that are interpreted during the execution of a Web service as they have been right before starting the execution. Finally, *out* denotes the objects in the information space. The symbols that have been added to Σ_0 can be used when formulating post-state constraints to describe changes between pre-states and post-states in a precise way.

Definition 4.1 (Abstract State Space). *An* ***abstract state space*** $\mathcal{A} = (\mathcal{S},\mathcal{U},\Sigma,\Omega,\omega)$ *is a 5-tuple such that*

(i) $\mathcal{S}$ *is a non-empty set of* ***abstract states****,*
(ii) $\mathcal{U}$ *is some non-empty set of objects called the* ***universe*** *of* $\mathcal{A}$
(iii) $\Omega \subseteq \mathcal{L}(\Sigma)$ *is consistent*
(iv) $\omega : \mathcal{S} \to Mod_\Sigma(\Omega, \mathcal{U}) \times \mathcal{P}(\mathcal{U})$ *is a total surjective function that assigns to every abstract state* s *a pair of a* Σ*-interpretation* $\omega_{rw}(s)$ *satisfying* Ω *and an information space* $\omega_{is}(s)$ *and*
(v) *for all* $s, s' \in \mathcal{S}$ *and* $\alpha \in \Sigma_S$: $meaning_{\omega_{rw}(s)}(\alpha) = meaning_{\omega_{rw}(s')}(\alpha)$.
□

In $\mathcal{A}$, Ω can be considered as a domain ontology representing (consistent) background knowledge about the world. It is used in any sort of descriptions, like preconditions etc. Clause (v) captures the nature of static symbols. In the following, $\mathcal{A}$ always denotes an abstract state space $\mathcal{A} = (\mathcal{S}, \mathcal{U}, \Sigma, \Omega, \omega)$.

For interacting with a Web service W, a client can use a technical interface. When abstracting from the technical details, every such interface basically provides a set of values as input data. The required input data represent the abstract interface used for interaction with the Web service from a capability point of view.

Definition 4.2 (Web Service Capability Interface, Input Binding). *A* ***Web service capability interface*** *IF of a Web service* W *is a finite sequences of names* $(i_1, \ldots, i_n)$ *of all required input values of a* W*. An* ***input binding*** β *for a Web service capability interface IF in* $\mathcal{A}$ *is a total function* $\beta : \{i_1, \ldots, i_n\} \to \mathcal{U}$*. The set of all input bindings for IF in* $\mathcal{A}$ *is denoted by* $In_{\mathcal{A}}(IF)$. □

An input binding essentially represents the input that is provided by the invoker of a Web service W during the *entire* execution of W.

Definition 4.3 (Web Service Execution). *A (possible)* ***Web service execution*** *in* $\mathcal{A}$ *is finite sequences* $\tau = (s_0, \ldots, s_m) \in \mathcal{S}^+$ *of abstract states such that for all* $0 \leq j < m$ *and* $0 \leq i, k \leq m$

(i) $\omega(s_j) \neq \omega(s_{j+1})$,
(ii) $\emptyset = \omega_{is}(s_0) \subseteq \omega_{is}(s_1) \subseteq \ldots \subseteq \omega_{is}(s_m)$,
(iii) $universe(\omega_{rw}(s_i)) = universe(\omega_{rw}(s_k))$,
(iv) $\omega_{is}(s_i) \subseteq universe(\omega_{rw}(s_i))$,
(v) *for all* $\alpha \in \Sigma_D$: $meaning_{\omega_{rw}(s_0)}(\alpha) = meaning_{\omega_{rw}(s_i)}(\alpha_{pre})$ *and*
(vi) $meaning_{\omega_{rw}(s_i)}(out) = \omega_{is}(s_i)$.

We denote the set of all possible Web service executions in $\mathcal{A}$ *by* $Exec(\mathcal{A})$.
□

This definition gives detailed conditions under which a sequence τ can be considered as a Web service execution. Clause (iii) requires that within an execution the universes which are related to abstract states s_j are the same[3]. In other words, universes (which are used to interpret state-based expression) that are related by an execution are not arbitrary, but specifically related to each other. In particular, (iii) ensures that within a functional description $\mathcal{D}$ postconditions can talk about *every object* that the precondition can refer to as well. Hence, precise comparisons between various states of an execution becomes possible. Clause (iv) requires that for every abstract state that involved in the execution its information space is part of the universe of the abstract state. This allows to relate and compare information space objects with real-world

[3]In order to model dynamic universes (e.g. object creation and deletion) one needs to model object existence in the state-description language $\mathcal{L}$ itself, for instance by a dynamic unary relation `existing`.

objects in state-based expressions. Finally, clauses (v) and (vi) ensure that in all intermediate and final states, the pre-versions α_{pre} of dynamic symbols α are interpreted as α in the prestate s_0 of the execution and that the symbol *out* represent the respective information space.

Definition 4.4 (Web Service, Web Service Implementation). *A **Web service implementation** W of some Web service capability interface $IF = (i_1, \ldots, i_n)$ in $\mathcal{A}$ is a total function $\iota : In_{\mathcal{A}}(IF) \times \mathcal{S} \rightarrow Exec(\mathcal{A})$ that defines for all accepted input bindings in $In_{\mathcal{A}}(IF)$ and abstract states $s \in \mathcal{S}$ the respective Web service execution of W in $Exec(\mathcal{A})$. Formally, we require for ι that $\iota(\beta, s) = (s_0, ..., s_m)$ implies $s_0 = s$ for all $s \in \mathcal{S}, \beta \in In_{\mathcal{A}}(IF)$. A **Web service** $W = (IF, \iota)$ is a pair of a Web service capability interface IF and a corresponding Web service implementation ι of IF.* □

One can consider the mapping ι as a marking of execution sequences in $\mathcal{A}$ by the input data that triggers the execution. Since we define a Web service implementation in terms of a function which maps to single Web service executions, we consider *deterministic* Web services, i.e the execution is fully determined by the input binding β and the intial state s_0 only. Any sort of uncertainty about what is going to happen when executing W (e.g. unexpected failures due to the environment the Web service is embedded in) is not considered in our model. In being a total function on $In_{\mathcal{A}}(IF) \times \mathcal{S}$, the definition reflects the fact that ι represents an (abstract) *implementation*, i.e. (unlike for specifications) every possible effect in every situation is *fully determined* by ι.

Based on this formal machinery, we can now formalize the meaning of functional descriptions $\mathcal{D} \in \mathcal{F}$ that are based on a state-description language $\mathcal{L}(\Sigma)$. In the following, we write $\mathcal{I}, \beta \models_{\mathcal{L}(\Sigma)} \phi$ to express that formula $\phi \in \mathcal{L}(\Sigma)$ is satisfied under Σ-interpretation $\mathcal{I}$ and variable assignment β. We assume that a functional description $\mathcal{D} = (\phi^{pre}, \phi^{post}, IF_{\mathcal{D}})$ consists of a precondition $\phi^{pre} \in \mathcal{L}(\Sigma_0)$, and a postcondition $\phi^{post} \in \mathcal{L}(\Sigma)$. $IF_{\mathcal{D}} \subseteq FreeVars(\phi^{pre}, \phi^{post})$ denotes the set of (free) variable names in $\mathcal{D}$ which represent inputs for the Web service under consideration. The logical expressions ϕ^{pre} and ϕ^{post} usually refer to some background ontology $\Omega \subseteq \mathcal{L}(\Sigma)$.

Definition 4.5 (Extension of an Input Binding). *Let β be an input binding for some Web service capability interface $IF = (i_1, \ldots, i_n)$, V be a set of symbol names and $U \subseteq \mathcal{U}$. A total function $\beta' : \{i_1, \ldots, i_n\} \cup V \rightarrow U$ is called a **V-extension of β in U** if $\beta'(i_j) = \beta(i_j)$ for all $1 \leq j \leq n$.* □

An extension of an input binding β is used in the next definition to ensure that every variable that occurs free in a precondition or postcondition can be assigned a concrete value. Otherwise, no truth-value can be determined for these statements.

Definition 4.6 (Renaming). *Let π be some function and β an input binding for IF. Then we denote by $rename_{\pi}(\beta)$ the input binding β' for IF' that is derived from β by replacing all pairs $(n, v) \in \beta$ with $n \in dom(\pi)$ by $(\pi(n), v)$. We call $rename_{\pi}(\beta)$ **renaming of β by π**.* □

The renaming represents on a technical level the effect of renaming input names in a Web service interface by the corresponding names in the interface used in the Web service description.

Definition 4.7 (Capability Satisfaction, Capability Model). *Let $W = (IF, \iota)$ be a Web service in $\mathcal{A}$ and $\mathcal{D} = (\phi^{pre}, \phi^{post}, IF_{\mathcal{D}})$ be a functional description of a Web service. Let FV denote the set of free variables in ϕ^{pre} and ϕ^{post} and U denote $universe(\omega_{rw}(s_0))$. W **satisfies capability** $\mathcal{D}$ in $\mathcal{A}$ if and only if*

(i) *there exists a subset* $IF' \subseteq IF_{\mathcal{D}}$ *of the inputs of* $\mathcal{D}$ *and a bijection* $\pi : IF \to IF'$ *between* IF *and* IF' *such that*

(ii) *for all input bindings* $\beta \in In_{\mathcal{A}}(IF)$ *and abstract states* $s \in \mathcal{S}$*: for all* FV*-extensions* β' *of* $rename_{\pi}(\beta)$ *in* U*: if* $\iota(\beta, s) = (s_0, \ldots, s_m)$ *for some* $m \geq 0$ *and* $\omega_{rw}(s_0), \beta' \models_{\mathcal{L}(\Sigma)} \phi^{pre}$ *then* $\omega_{rw}(s_m), \beta' \models_{\mathcal{L}(\Sigma)} \phi^{post}$

In this case we write $W \models_{\mathcal{F}} \mathcal{D}$ *and call the Web service* W *a* ***capability model*** *(or simply model) of* $\mathcal{D}$ *in* $\mathcal{A}$. □

Clause (i) essentially requires (interface) compatibility between the Web service and the inputs refered to in Web service description. Note, that we do not require syntactic equality between these names, but only equivalence up to some renaming π. Moreover, it is perfectly fine for models of $\mathcal{D}$ to only use a proper subset IF' of the inputs $IF_{\mathcal{D}}$ mentioned in capability $\mathcal{D}$. Clause (ii) defines the meaning of preconditions and postcondition. Please note, that free variables in these expressions are implicitly universally quantified by our definition. Futhermore, Def. 4.3 ensures that universes do not change during the execution from s_0 to s_m (i.e. $universe(\omega_{rw}(s_m)) = U$) and therefore, β' is a valid variable assignment not only for $\omega_{rw}(s_0)$, but for the interpretation $\omega_{rw}(s_m)$ as well.

The reason why we need to use an arbitrary bijection π in the definition (instead of the specific one id, i.e. the identity on input names) is as follows: Consider some Web Service $W = (I, \iota)$ that provides a particular capability $\mathcal{D}$, i.e. $W \models_{\mathcal{F}} \mathcal{F}$. Now imagine that we change the Web Service Interface I to an interface I' by simply renaming all input names and that we adapt the implementation ι to ι' to use these new input names respectively. Clearly, this way we create another Web Service $W' = (I', \iota') \neq W$ that precisely does the same as W. We would expect that in this case $W' \models_{\mathcal{F}} \mathcal{D}$ as well. However, $W' \not\models_{\mathcal{F}} \mathcal{D}$ because of incompatible (syntactic) interfaces, which would result in too restrictive definition.

4.3 Applying the formal Model for Semantic Analysis

For demonstrating the suitability of the proposed model, this section shows its beneficial application for semantic analysis of functional descriptions. Based on our model-theoretic framework, we can carry over several semantic standard notions from mathematical logic [Enderton, 2000; Fitting, 1996] that refer to formal descriptions and are based on the *model* notion to our particular context in a meaningful way.

4.3.1 Realizability

We define *realizability* of a description $\mathcal{D}$ as the corresponding notion to satisfiability in a logic $\mathcal{L}$:

Definition 4.8 (Realizability). *Let* $\mathcal{A}$ *be an abstract state space. A functional description* $\mathcal{D}$ *is* ***realizable in*** $\mathcal{A}$ *iff. there is a Web service* W *in* $\mathcal{A}$ *that satisfies* $\mathcal{D}$*, i.e.* $W \models_{\mathcal{F}} \mathcal{D}$. □

Consider the following functional description $\mathcal{D} = (\phi^{pre}, \phi^{post}, IF_{\mathcal{D}})$ describing Web services for account withdraws: $IF_{\mathcal{D}} = \{?acc, ?amt\}$

$$\phi^{pre} : ?amt \geq 0 \qquad \phi^{post} : balance(?acc) = balance_{pre}(?acc) - ?amt$$

At a first glance, the given description seems to be implementable within some Web service W that satisfies $\mathcal{D}$. However, taking a closer look at the respective domain ontology it becomes obvious that this actually is not the case. The ontology defines that a balance might not be negative, but the precondition does not prevent the balance being less then the withdraw. Let's assume that there is a Web service W realizing $\mathcal{D}$. When considering an input binding β with $\beta(?amt) > balance_{pre}(?acc)$, then the precondition is satisfied and thus the postcondition should hold in the final state of the respective execution, i.e. $\omega_{rw}(s_m), \beta \models \forall ?acc.balance(?acc) < 0$. However, this is inconsistent with the domain ontology since $\Omega \models balance(?acc) \geq 0$ and thus s_m can not exist in $\mathcal{A}$. This is a contradiction and shows that no Web service W with $W \models_{\mathcal{F}} \mathcal{D}$ can exist. To fix the description such that it becomes realizable, we need to extend the precondition to $\phi^{pre} : 0 \leq ?amt \wedge ?amt \leq balance(?acc)$.

The example illustrates the usefulness of the notion of realizability. It provides a tool for detecting functional descriptions that contain flaws that might not be obvious to the modelers. Moreover, we as we will see soon, we can often rephrase the problem of realizability of a description $\mathcal{D} \in \mathcal{F}$ to a well-understood problem in $\mathcal{L}$ for which algorithms already exist. We first turn to an important other notion of which realizability turns out to be a special case (in conformance as with the original notions in mathematical logic).

4.3.2 Functional Refinement

The notion of logical entailment is usually defined as follows: An formula ϕ logically entails a formula ψ iff every interpretation $\mathcal{I}$ which is a models of ϕ (i.e. $\mathcal{I} \models_{\mathcal{L}} \phi$) is also a model of ψ. Substituting interpretations by Web services, formulae by functional descriptions and the satisfaction $\models_{\mathcal{L}}$ by capability satisfaction $\models_{\mathcal{F}}$ we derive a criteria that captures *fuctional refinement*:

Definition 4.9 (Functional Refinement). *Let $\mathcal{A}$ be an abstract state space and $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{F}$ be functional descriptions. $\mathcal{D}_1$ is a **functional refinement of** $\mathcal{D}_2$ **in** $\mathcal{A}$ iff. for each Web service W in $\mathcal{A}$, $W \models_{\mathcal{F}} \mathcal{D}_1$ implies $W \models_{\mathcal{F}} \mathcal{D}_2$.*
We use $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ to denote that description $\mathcal{D}_1$ is a functional refinement of description $\mathcal{D}_2$ in $\mathcal{A}$. □

Intuitively speaking, $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ means that $\mathcal{D}_1$ is more specific than $\mathcal{D}_2$: Every Web service (no matter which one) that provides $\mathcal{D}_1$ can also provide $\mathcal{D}_2$. In other words, $\mathcal{D}_1$ must describe some piece of functionality that always fits the requirements $\mathcal{D}_2$ as well. However, Web services that provide $\mathcal{D}_2$ do not have to satisfy $\mathcal{D}_1$ and therefore, a Web service that provides $\mathcal{D}_1$ can do something more specific than required by $\mathcal{D}_2$.

For illustration, consider some Web service description $\mathcal{D}_1 = (\phi_1^{pre}, \phi_1^{post}, IF_1)$ with $IF_1 = \{?prs, ?acc\}$ that advertises the ability to provide access credentials for a particular web site (*http://theSolution.com*). A domain ontology specifies that if some web site has some content and someone can access the web site, then he (is able to) know about the content. Furthermore, *http://theSolution.com* is a web site providing the ultimate answer to life (the universe and everything) and some constant $accessFee$ has a value less then 42.[4]

[4]Note that we do not expect such knowledge in one central domain ontology, but a number of knowledge bases (generic, provider- and requester-specific). For simplicity we assume Ω being already aggregated

$$\phi_1^{pre} : account(?p, ?acc) \wedge balance(?acc) \geq accessFee$$
$$\phi_1^{post} : balance(?acc) = balance_{pre}(?acc) - accessFee$$
$$\wedge \mathbf{out}(password(?prs, http://theSolution.com))$$
$$\wedge isValid(password(?prs, http://theSolution.com))$$
$$\Omega \models \forall ?ws, ?co, ?prs.\ content(?ws, ?co) \wedge access(?prs, ?ws) \Rightarrow knows(?prs, ?co)$$
$$content(http://theSolution.com, answer2Life), accessFee \leq 42$$
$$\forall ?prs, ?ws.\ isValid(password(?prs, ?ws)) \Rightarrow access(?prs, ?ws))$$

Using our formal definition we now can examine another definition $\mathcal{D}_2 = (\phi_2^{pre}, \phi_2^{post}, IF_2)$ with $IF_2 = \{?prs, ?acc\}$ and check if it is a functional refinement of the previous description.

$$\phi_2^{pre} : account(?prs, ?acc) \wedge balance(?acc) \geq 100 \quad \phi_2^{post} : knows(?prs, answer2Life)$$

This notion can beneficially be applied within functionality-based matchmaking. For instance, let's assume that a Person me is seeking for the ultimate answer to life ($knows(me, answer2Life)$); me has an account $acc123$ with a current balance of 174 USD. Given this information (and our domain ontology Ω) and considering the specific input binding $\beta(?prs) = me, \beta(?acc) = acc123$, we can infer that any Web service W that is advertised to provide capability $\mathcal{D}_2$ can serve for me's purpose as the precondition ϕ_2^{pre} is satisfied for the input β. In consequence, for the specific input β the service delivers what is described the postcondition ϕ_2^{post}; therefrom, we can infer $knows(me, answer2Life)$. However, since $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ we know as well, that any Web service W' that is advertised to provide capability $\mathcal{D}_1$ is perfectly suitable for me and his endeavor as well. The notion of functional refinement can then be used to pre-index some set of Web service description, such that for a given request it is not necessary to consider all available description but only a subset identified by the pre-indexing.

We can extend the notion of functional refinement to the notion of *functional equivalence* of two descriptions as follows:

Definition 4.10 (Functional Equivalence). *Let $\mathcal{A}$ be an abstract state space. Two functional descriptions $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{F}$ are **functionally equivalent in** $\mathcal{A}$ iff. $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ and $\mathcal{D}_2 \sqsubseteq \mathcal{D}_1$.*
We use $\mathcal{D}_1 \equiv \mathcal{D}_2$ to denote that $\mathcal{D}_1$ and $\mathcal{D}_2$ are functionally equivalent descriptions in $\mathcal{A}$. □

Clearly, both notions refinement and equivalence are relevant in the context of functional matchmaking and Web service discovery.

Our framework allows to proof the following theorem (see Appendix A), which is especially useful for reducing the problem of determining functional refinement (and eventually all other semantic analysis notions we discuss in this section) to a well-defined proof obligation in the language $\mathcal{L}$ underlying $\mathcal{F}$.

Theorem 4.1 (Reduction of Functional Refinement from $\mathcal{F}$ to $\mathcal{L}$, Sufficient Condition, Interface Identify). *Let $\mathcal{D}_1 = (\phi_1^{pre}, \phi_1^{post}, IF_1)$ and $\mathcal{D}_2 = (\phi_2^{pre}, \phi_2^{post}, IF_2)$ be functional descriptions in $\mathcal{F}$ with the same interfaces, i.e. $IF_1 = IF_2$. Let $[\phi]_{\Sigma_D^{pre} \to \Sigma_D}$ denote the formula ϕ' which can be derived from ϕ by replacing any dynamic symbol $\alpha \in \Sigma_D$ by its corresponding pre-variant $\alpha_{pre} \in \Sigma_D^{pre}$. Let $Cl_\forall(\phi)$ denote the universal closure of formula ϕ, i.e. a closed formula ϕ' of the form $\forall x_1, \ldots, x_n(\phi)$ where $x_1, \ldots, x_n$ are all the variable that occurs free in ϕ. Let the entailment relation $\models_\mathcal{L}$ be defined in model-theoretic terms, i.e. $\phi_1 \models_\mathcal{L} \phi_2$ iff. for any interpretation I that satisfies*

ϕ_1 *it holds* I *satisfies* ϕ_2 *too.*

Then $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ *if*

(i) $\Omega \models_{\mathcal{L}} Cl_\forall(\phi_2^{pre} \Rightarrow \phi_1^{pre})$ *and*

(ii) $\Omega \cup [\Omega]_{\Sigma_D^{pre} \to \Sigma_D} \models_{\mathcal{L}} Cl_\forall(([\phi_2^{pre}]_{\Sigma_D^{pre} \to \Sigma_D} \wedge \phi_1^{post}) \Rightarrow \phi_2^{post})$

□

This gives us the following: If there is an algorithm or an implemented system that allows us to determine logical entailment in $\mathcal{L}$, then we can use the very same system or algorithm to determine[5] functional refinement for descriptions of the capability language $\mathcal{F}$, i.e. no new calculus for dealing with $\mathcal{F}$ is needed (at least for the purpose semantic analysis). However, the algorithm which can be derived from Theorem 4.1 is no longer a heuristic, but *provably correct.*

A note on $[\Omega]_{\Sigma_D^{pre} \to \Sigma_D}$**.** Some discussion on efficiency of the reduction method proposed by Theorem 4.1 are in order: a close inspection of the proof of Theorem 4.1 that is given in Appendix A shows that it is not actually required to rename *all* dynamic symbols in Ω (and thus to use $[\Omega]_{\Sigma_D^{pre} \to \Sigma_D}$ in the criterion on the righthand side) but only those that occur in the postconditions. Hence, instead of $[\Omega]_{\Sigma_D^{pre} \to \Sigma_D}$ we can use a significantly smaller set of formula that we add to the orginal ontologies Ω in Theorem 4.1 without violating its validity. The ratio of dynamic symbols in Ω is expected to be significantly less than the ration of static symbols in Ω. For these two reasons, we do not expect problems with the renaming of Ω in Clause (ii) of the theorem when the size of Ω is big. ◁

Related Results. In [Zaremski and Wing, 1997], functional matching of software components has been discussed in detail and a similar criterion to the one we developed in Theorem 4.1 has been proposed in the context of component specification matching under the term *Guarded Plugin Match*:

$$match_{guarded-pm}(\mathcal{D}_1, \mathcal{D}_2) = (\phi_2^{pre} \Rightarrow \phi_1^{pre}) \wedge (\phi_1^{pre} \wedge \phi_1^{post} \Rightarrow \phi_2^{post})$$

However, [Zaremski and Wing, 1997] covers a simpler scenario, where specifications do not contain *dynamic* functions.

In this respect, our result can be seen as a generalization of the notion of *Guarded Plugin Match* to a context where certain properties of the world can be persistenly changed during the use of a software component. Futhermore, our criterion explicitly deals with a background ontology Ω on which the functional descriptions $\mathcal{D}_1, \mathcal{D}_2$ are based.

In contrast to Theorem 4.1, [Zaremski and Wing, 1997] gives no formal investigation of how the criterion called *Guarded Plugin Match* actually relates to the notion of functional refinement. ◁

[5]Since Theorem 4.1 is only a sufficient condition, we actually can not really speak of a reduction in the sense of the theory of computation or the theory of computational complexity. More precicely, we have a partial reduction, in the sense that for some (and perhaps most cases in practice) the proofobligations mentioned on the righthand side in the theorem allow us to determine that functional refinement holds. However, there might be cases where functional refinements hold, but the criterion is not met. Currently, we are working on finding a theorem that explains the reverse direction of Theorem 4.1, i.e. a theorem that gives a necessary condition for functional refinement. We expect that there is such a theorem, since in the case of functional refinement between two capability descriptions $\mathcal{D}_1, \mathcal{D}_2$, the respective elements of which $\mathcal{D}_1, \mathcal{D}_2$ consists can not have an arbitray relationship.

Requirement on the definition of $\models_{\mathcal{L}}$. Theorem 4.1 actually makes an assumption on the definition of logical entailment in the state-description language $\mathcal{L}$, i.e. $\models_{\mathcal{L}}$ is required to be defined in the standard model-theoretic way. The proof of the theorem which is give in Appendix A, actually uses this requirement.

In practice, this requirement should not be considered as a severe restriction, since for many logics, especially the ones that are interesting for our purposes, this requirement is met, e.g. (classical) First-order Logics, Description Logics. However, there are logics for which the entailment relation is *not* defined in terms of model-theory, but in terms of proof-theory, i.e. a formula ϕ_2 can is entailed by another formula ϕ_1 iff. certain proof rules can be used to derive ϕ_2 from ϕ_1.

A specific example of such logics are paraconsistent logics like the ones discussed in [Hunter, 1998]: Since the standard definition of entailment in model-theoretic terms is explosive, paraconsistent logics often weaken entailment of a (non-paraconsistent) system $\mathcal{L}$ by defining entailment in proof-theoretic terms and disallowing certain inferences that are sound under the standard model-theoretic definition of $\models_{\mathcal{L}}$, for instance *ex-falso-quodlibet*. ◁

Generalizing Theorem 4.1. In fact, it is possible to weaken the assumption on the interfaces IF_1, IF_2 in the theorem to a much more general case: Instead of requiring the interfaces to be the same, we can consider interfaces IF_1, IF_2 that are different but where some bijection $\pi : IF_1 \to IF_2$ exists, such that after normalization of the descriptions by renaming input names (based on the mapping π, the original criterion mentioned in Theorem 4.1 is met.

For instance, consider two descriptions $\mathcal{D}_1, \mathcal{D}_2$ which advertise a Web Service that can compute the sum of two given input values. However, in $\mathcal{D}_1$ opA and opB are used to denote the inputs, wher eas $summandLeft$ and $summandRight$ are used.

For this case, we can easily construct a more general theorem that can effectively be proven by reducing the problem to renaming (i.e applying the bijection) and using Theorem 4.1. This way, we do not have to deal with the additional complexity in the proof of Theorem 4.2 itself:

Theorem 4.2 (Reduction of Functional Refinement from $\mathcal{F}$ to $\mathcal{L}$, Sufficient Condition, Interface Identify modulo Input Renaming). *Let $\mathcal{D}_1 = (\phi_1^{pre}, \phi_1^{post}, IF_1)$ and $\mathcal{D}_2 = (\phi_2^{pre}, \phi_2^{post}, IF_2)$ be functional descriptions in $\mathcal{F}$.*

Let $[\phi]_{\Sigma_D^{pre} \to \Sigma_D}$ denote the formula ϕ' which can be derived from ϕ by replacing any dynamic symbol $\alpha \in \Sigma_D$ by its corresponding pre-variant $\alpha_{pre} \in \Sigma_D^{pre}$. Let $Cl_\forall(\phi)$ denote the universal closure of formula ϕ, i.e. a closed formula ϕ' of the form $\forall x_1, \ldots, x_n(\phi)$ where $x_1, \ldots, x_n$ are all the variable that occurs free in ϕ. Let $[\phi]_\pi$ denote the formula ϕ' which can be derived from ϕ by replacing any free occurence of a variable $x \in IF_1$ by the corresponding name $\pi(x) \in IF_2$, where $\pi : IF_1 \to IF_2$ is a mapping between IF_1 and IF_2. Let the entailment relation $\models_{\mathcal{L}}$ be defined in model-theoretic terms, i.e. $\phi_1 \models_{\mathcal{L}} \phi_2$ iff. for any interpretation I that satisfies ϕ_1 it holds I satisfies ϕ_2 too.

Then $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ if there exists a bijection $\pi : IF_1 \to IF_2$ such that

(i) $\Omega \models_{\mathcal{L}} Cl_\forall(\phi_2^{pre} \Rightarrow [\phi_1^{pre}]_\pi)$ *and*

(ii) $\Omega \cup [\Omega]_{\Sigma_D^{pre} \to \Sigma_D} \models_{\mathcal{L}} Cl_\forall(([\phi_2^{pre}]_{\Sigma_D^{pre} \to \Sigma_D} \wedge [\phi_1^{post}]_\pi) \Rightarrow \phi_2^{post})$

□

Clearly, the computational complexity of determining functional refinement based in our reduction increases in the more general case with respect to the simple case of interface identity: Assume there is a bijection π between the interfaces. Then the number of input names in IF_1 and IF_2 are the same. Let n denote this number of input variables. Then, there are $n!$ different bijections between interfaces IF_1 and IF_2. Thus, in the worst case the computational cost of the more general criterion of Theorem 4.2 grows exponentially in the size of the interfaces of functional descriptions $\mathcal{D}_1$ and $\mathcal{D}_2$.

Realizablity vs. Refinement. To be able to formulate the next corollary (which is an immediate consequence of the definition of realizability and functional refinement), we use $\bot^{IF}$ to denote a description $D \in \mathcal{F}$ that is trivially unrealizable, i.e. $D = (true, false, IF)$.

Corollary 4.1 (Realizability vs. Refinement). *A functional description* $\mathcal{D} = (\phi^{pre}, \phi^{post}, IF)$ *is not realizable iff.* $\mathcal{D} \sqsubseteq \bot^{IF}$ □

The corollary simply states that any description which is more specific than the trivially unrealizable functional description must be unrealizable as well. In the light of Theorem 4.1, it shows that we can reduce realizability of $\mathcal{D}$ to a well-defined proof obligation in $\mathcal{L}$ as well. Hence we can deal with realizability algorithmically based on existing tools.

4.3.3 Omnipotence

For any functional description $\mathcal{D}$ we can consider the dual notion of being not realizable at all, i.e. having every Web service W in $\mathcal{A}$ as a model. This notion corresponds to the classical notion of validity and obviously represents another form of ill-defined or unacceptable type of description. It matches all possible Web services, no matter what they actually do. Service providers could use such (non-trivially) omnipotent descriptions to advertise their Web services in some registry to get maximal visibility. A trivially omnipotent functional description in $\mathcal{F}$ is $\top^{IF} = (true, true, IF)$.

As an immediate consequence we can derive the following corollary which shows that we can reduce omnipotence of $\mathcal{D}$ to a well-defined proof obligation in $\mathcal{L}$ as well and thus deal with it algorithmically based on existing tools:

Corollary 4.2 (Omnipotence vs. Refinement). *A functional description* $\mathcal{D} = (\phi^{pre}, \phi^{post}, IF)$ *is omnipotent iff.* $\top^{IF} \sqsubseteq \mathcal{D}$ □

The corollary simply states that any description which is more general than the trivially omnipotent functional description must be omnipotent as well.

Summary. Semantic analysis can be seen as both, (i) a concrete example of symbolic computation with functional descriptions that we can formally ground in our formal model, and (ii) as a problem that is interesting in itself. Using our model, we are able to rigorously define various useful notions that enable us to analyze and relate functional descriptions semantically. We have shown that we can reduce the various relevant notions to well-defined proof obligations in the underlying language $\mathcal{L}$ without making severe restrictions or assumptions on that language. Using our framework, we are able to proof the correctness of the reduction. Given the a wealth of different languages that co-exist on the Semantic Web (and the ones that might still be invented), our uniform treatment provides a universal approach to the semantics of functional description independent of the language used.

4.4 Extensions of Basic Functional Descriptions

So far our model of functional descriptions (i.e. the functional description language $\mathcal{F}$) only uses desription elements that can be considered as standard elements of frameworks for the semantic descriptions of software components, namely preconditions and postconditions.

In the following, we will briefly discuss a few extensions to these basic elements of functional descriptions (and thus the functional description language $\mathcal{F}$) that allow our framework to be applied in a wider range of application scenarios and simplify its usage in concrete applications. Hereby, some extensions go beyond the expressivity of the basic framework, whereas others simply provide convenience for modelers by enabling to write more concise descriptions of Web service capabilities.

4.4.1 Execution Invariants

The classical perspective taken on computational entities in most existing frameworks for functional specification are based on a black-box perspective: A computational entity (e.g. function, method, Web service etc.) is considered as a black-box with an interface, where an invocation (by providing some input data) in some valid pre-state (captured by a precondition) leads to an execution that ends in some post-state (characterized by a postcondition) of the execution. However, all executions are considered as *atomic* state-changes, i.e. no intermediary states are visible to the observer of an execution. Consequently, functional descriptions do not refer[6] to intermediary states, as it is for example possible in Transaction Logic [Bonner and Kifer, 1995].

On the other hand, this means that we can not express properties that are guaranteed to hold *always* during the execution of a computational entity, as it might be interesting for security or safety related properties, and in the context of the composition of a system by orchestration of various computational components, for instance Web Services.

For such kind of application scenarios, we present an extension of our basic model (that is applied in the context of the WSMO intitiative as well, see for instance [Lausen et al., 2005]) that allows to state properties in the functional specification which are guaranteed to hold at every point in time during the execution of the Web Service, and not only in the pre-state or the post-state of the execution. Subsequently, we will refer to such properties as *execution invariants*. In the following, we will show how to extend the syntax and semantics of our functional description language $\mathcal{F}$ accordingly. Finally, we will briefly discuss, how these extensions affect our discussion of the Semantic Analysis of functional descriptions and give an extended version of Theorem 4.2.

Extending the Syntax of $\mathcal{F}$. So far, we assumed a functional description $\mathcal{D} \in \mathcal{F}$ to be a triple $\mathcal{D} = (\phi^{pre}, \phi^{post}, IF)$ consisting of a precondition ϕ^{pre}, a postcondition ϕ^{post} and a interface definition IF. Now, we extend the triple by an additional element that we call *execution invariant*: a (state-related) formula $\phi^{inv} \in \mathcal{L}(\Sigma)$. Since we allow formulas over the extended signature $\Sigma \supseteq \Sigma_0$, it is possible in particular, to refer to the interpretation of dynamic symbols α at the beginning of the execution (by means of the respective pre-variant α_{pre}).

[6] This means both, either directly by means of names for states or indirectly by conditions that these intermediary states need to satisfy.

Definition 4.11 (Functional Description with Execution Invariants). *A **functional description $\mathcal{D}$ with execution invariants** is a quadruple*

$$\mathcal{D} = (\phi^{pre}, \phi^{post}, \phi^{inv}, IF)$$

where $\phi^{pre} \in \mathcal{L}(\Sigma_0)$ is called the precondition *of $\mathcal{D}$, $\phi^{post} \in \mathcal{L}(\Sigma)$ is called the* postcondition *of $\mathcal{D}$, , $\phi^{inv} \in \mathcal{L}(\Sigma)$ is called the* execution invariant *of $\mathcal{D}$ and $IF \subseteq FreeVars(\phi^{pre}, \phi^{post}, \phi^{inv})$ is called the capability interface. We denote the language of all functional descriptions with execution invariants by $\mathcal{F}_{inv}$.* □

Extending the Semantics of $\mathcal{F}$. Intuitively, an execution invariant ϕ^{inv} represents a property (of some *state*) that is never violated during the execution of a Web Service W. More precisely, in all intermediate states s_j of any execution of W, we require ϕ^{inv} to be satisfied. We have various options for our definition: First, do we only consider intermediate states for our definition or do we consider the whole executions (including the pre-state s_0 and final state s_m)? We decide to use only the intermediate states, since this gives us the most freedom: Satisfaction in pre-states can be enforced by adding ϕ^{inv} conjunctively to the precondition and satisfaction in post-states can be expressed by adding ϕ^{inv} conjunctively to the postcondition. We can thus cover all situations that can be expressed when considering all the states in a Web Service execution for the definition of the satisfaction of execution invariants. Furthermore, we can express situations that can not be covered by such a definition. Second, we can consider execution guarantees to be fulfilled when the intermediate states of any Web Service execution respects ϕ^{inv}, or use a bit weaker condition, that only considers the executions starting in a state s_0 that satisfies the precondition (wrt. the given input). Since, we share the perspective of all common software specification frameworks, that a violation of a precondition, somehow represents a illegal invocation of a computational entity, we decide to use the second approach here.

We will now adapt our definitions for describing the semantics of functional descriptions in $\mathcal{F}$ from above, to the extended language $\mathcal{F}_{inv}$. More precisely, we only need to adapt Definition 4.7 slightly:

Definition 4.12 (Capability Satisfaction, Capability Model in $\mathcal{F}_{inv}$). *Let $W = (IF, \iota)$ be a Web service in $\mathcal{A}$ and $\mathcal{D} = (\phi^{pre}, \phi^{post}, \phi^{inv} IF_{\mathcal{D}}) \in \mathcal{F}_{inv}$ be a functional description of a Web service with execution invariants. Let FV denote the set of free variables occuring in ϕ^{pre}, ϕ^{post} and ϕ^{inv} and let U denote $universe(\omega_{rw}(s_0))$.*

*W **satisfies capability** $\mathcal{D}$ in $\mathcal{A}$ if and only if*

- *(i)* *there exists a subset $IF' \subseteq IF_{\mathcal{D}}$ of the inputs of $\mathcal{D}$ and a bijection $\pi : IF \to IF'$ between IF and IF' such that*
- *(ii)* *for all input bindings $\beta \in In_{\mathcal{A}}(IF)$ and abstract states $s \in \mathcal{S}$: for all FV-extensions β' of $rename_{\pi}(\beta)$ in U: if $\iota(\beta, s) = (s_0, \ldots, s_m)$ for some $m \geq 0$ and $\omega_{rw}(s_0), \beta' \models_{\mathcal{L}(\Sigma)} \phi^{pre}$ then $\omega_{rw}(s_m), \beta' \models_{\mathcal{L}(\Sigma)} \phi^{post}$ and*
- *(iii)* *for all input bindings $\beta \in In_{\mathcal{A}}(IF)$ and abstract states $s \in \mathcal{S}$: for all FV-extensions β' of $rename_{\pi}(\beta)$ in U: if $\iota(\beta, s) = (s_0, \ldots, s_m)$ for some $m \geq 0$ and $\omega_{rw}(s_0), \beta' \models_{\mathcal{L}(\Sigma)} \phi^{pre}$ then $\omega_{rw}(s_j), \beta' \models_{\mathcal{L}(\Sigma)} \phi^{inv}$ for any $0 < j < m$*

*In this case we write $W \models_{\mathcal{F}_{inv}} \mathcal{D}$ and call the Web service W a **capability model** (or simply model) of $\mathcal{D}$ in $\mathcal{A}$.* □

Clauses (i) and (ii) are precisely the same as in Definition 4.7 for our language $\mathcal{F}$. Clearly, it is possible to write the same definition more concisely by merging

clauses (ii) and (iii). However, we skipped this merging to make clear that we face an orthogonal extension of $\mathcal{F}$ which results in an orthogonal extension of the *satisfaction* notion for capability descriptions.

According to our new definition of satisfiability in $\mathcal{F}_{inv}$, we get the following relationship between $\mathcal{F}$ and $\mathcal{F}_{inv}$ which allows us to consider $\mathcal{F}$ as a special case of $\mathcal{F}_{inv}$: For every description $\mathcal{D} = (\phi^{pre}, \phi^{post}, IF) \in \mathcal{F}$ there is a description $\mathcal{D}' \in \mathcal{F}_{inv}$ which is equivalent in any abstract state space $\mathcal{A}$, i.e. a description $\mathcal{D}'$ such that $W \models_{\mathcal{F}} \mathcal{D}$ iff. $W \models_{\mathcal{F}_{inv}} \mathcal{D}'$ for all Web Services W in $\mathcal{A}$ holds. In particular, $\mathcal{D}' = (\phi^{pre}, \phi^{post}, true, IF)$ has this property.

Extending the Semantic Analysis of Descriptions in $\mathcal{F}$. As we have seen in Section 4.3, for us the central notion in Semantic Analysis is *functional refinement*, since every other considered notion can be reduced to this notion. We have demonstrated by means of Theorems 4.1 and 4.2 that we can reduce functional refinement (which is a notion in $\mathcal{F}$) to logical entailment in underlying language $\mathcal{L}$ for describing states. In the following, we give a theorem for functional refinement in the extended language $\mathcal{F}_{inv}$ that is analogous to Theorem 4.1.

Theorem 4.3 (Reduction of Functional Refinement from $\mathcal{F}_{inv}$ to $\mathcal{L}$, Sufficient Condition, Interface Identify). *Let $\mathcal{D}_1 = (\phi_1^{pre}, \phi_1^{post}, \phi_1^{inv}, IF_1)$ and $\mathcal{D}_2 = (\phi_2^{pre}, \phi_2^{post}, \phi_2^{inv}, IF_2)$ be functional descriptions in $\mathcal{F}_{inv}$ with the same interfaces, i.e. $IF_1 = IF_2$. Let $[\phi]_{\Sigma_D^{pre} \to \Sigma_D}$ denote the formula ϕ' which can be derived from ϕ by replacing any dynamic symbol $\alpha \in \Sigma_D$ by its corresponding pre-variant $\alpha_{pre} \in \Sigma_D^{pre}$. Let $Cl_\forall(\phi)$ denote the universal closure of formula ϕ, i.e. a closed formula ϕ' of the form $\forall x_1, \ldots, x_n(\phi)$ where $x_1, \ldots, x_n$ are all the variable that occurs free in ϕ. Let the entailment relation $\models_{\mathcal{L}}$ be defined in model-theoretic terms, i.e. $\phi_1 \models_{\mathcal{L}} \phi_2$ iff. for any interpretation I that satisfies ϕ_1 it holds I satisfies ϕ_2 too.*

Then $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ if

(i) $\Omega \models_{\mathcal{L}} Cl_\forall(\phi_2^{pre} \Rightarrow \phi_1^{pre})$ *and*

(ii) $\Omega \cup [\Omega]_{\Sigma_D^{pre} \to \Sigma_D} \models_{\mathcal{L}} Cl_\forall(([\phi_2^{pre}]_{\Sigma_D^{pre} \to \Sigma_D} \wedge \phi_1^{inv}) \Rightarrow \phi_2^{inv})$ *and*

(iii) $\Omega \cup [\Omega]_{\Sigma_D^{pre} \to \Sigma_D} \models_{\mathcal{L}} Cl_\forall(([\phi_2^{pre}]_{\Sigma_D^{pre} \to \Sigma_D} \wedge \phi_1^{post}) \Rightarrow \phi_2^{post})$

□

In the same way, we can adapt the more general Theorem 4.2 for the extended language $\mathcal{F}_{inv}$.

Conclusion. The proposed extension allows a modeler to reveal detailed information that is global to all intermediate states of any execution of a Web Service. This information might be useful in various specific application scenarios of functional descriptions of Web services.

However, execution guarantees as we introduced them here are less general and less expressive than general languages for characterizing state-sequences, for instance Temporal Logics and Transaction Logics. Since our goal still is to keep the overhead added by $\mathcal{F}$ on top of $\mathcal{L}$ limited (for instance by using the same language $\mathcal{L}$ to describe execution guarantees and not introducing a separate language for these elements of capabilities), we do not aim at a more general language $\mathcal{F}$ here at present.

4.4.2 Complete and Incomplete Descriptions

A common problem in specification of the functionality of computational entities that can alter the state of the world is the so-called *Frame problem*: Descriptions usually describe positive information about what changes happen, however, to keep descriptions simple and manageable, they do not explicitly state what does not happen[7]. Incomplete descriptions can pose difficulties when processing such descriptions for some higher-level task such as Web Service discovery, since they are weaker (i.e. contain less information) as complete descriptions.

Complete descriptions can be achieved by adding supplementary descriptions about all things that stay the same during the execution. Given a fixed model of the world, such a completion process can be automated by adding so-called frame axioms to the existing (incomplete) description. However, the manual addition of such axioms is a tedious task for humans and should be avoided.

There are two options that one can take to resolve this problem: First, one can decide to interpret all functional descriptions as being complete, and generating the frame-axioms automatically. This relieves the modeler from writing larger specifications, but has the drawback that it is no longer possible to write down incomplete descriptions on purpose (for instance, to express vague knowledge or minimal guarantees about the behavior of some computational entity). Second, one can allow the modeler to express *explicitly* as part of the description whether the description is complete or not by means of a keyword. This way, we avoid the drawback of the first approach, while keeping simplicity of descriptions for modelers.

Functional descriptions consist of various independent elements, which are descriptions (of states) themselves. Again, we have some freedom in our approach, that we need to decide upon: First, we can mark a capability description as a whole as being complete or incomplete. Second, we can mark the single elements individually as being complete or incomplete. We will take the second approach, since it is more fine-grained. As we will see in the next paragraph, for the basic language $\mathcal{F}$ indeed both alternatives are the same.

As we explained above, completeness and incompleteness is relevant for all descriptions that are related to *what* happens (or alternatively, what does not). This is the case for postconditions (as well as executional invariants in the case of $\mathcal{F}_{inv}$). However, these properties are *not* relevant to description elements that specify *when* (more precisely: under what circumstances) something happens, i.e. preconditions. Consequently, we allow a modeler in the following to mark postconditions and other similar description elements.

In the following, we will show how to extend our basic capability description language $\mathcal{F}$ by means to distinguish complete and incomplete descriptions. We then give semantics to the extended functional descriptions. Finally, we discuss the effects of the extension on Semantic Analysis and give some conclusions.

Extending the Syntax of $\mathcal{F}$.

Definition 4.13 (Functional Description with Complete and Incomplete Elements). *A **functional description $\mathcal{D}$ with complete and incomplete elements** is a triple*

$$\mathcal{D} = (\phi^{pre}, (\phi^{post}, \gamma^{post}), IF)$$

[7] In particular, one can expect that there are substantially many more things that *don't happen* during the execution of a Web Service, than things that actually occur and can be observed.

where $\phi^{pre} \in \mathcal{L}(\Sigma_0)$ *is called the* precondition *of* $\mathcal{D}$, $\phi^{post} \in \mathcal{L}(\Sigma)$ *is called the* postcondition *of* $\mathcal{D}$, $\gamma^{post} \in \{\texttt{complete}, \texttt{incomplete}\}$ *is the* completeness marker *of* ϕ^{post} *and indicates whether the postcondition is considered as being complete or incomplete,* $IF \subseteq FreeVars(\phi^{pre}, \phi^{post})$ *is called the capability interface. We denote the language of all functional descriptions with complete and incomplete elements* $\mathcal{F}_{comp}$. □

Extending the Semantics of $\mathcal{F}$. For incomplete descriptions $\mathcal{D}$, the semantics is precisely the same as before (i.e. in our basic language $\mathcal{F}$), i.e. only information that is explicitly represented in the description elements ϕ^{pre} and ϕ^{post} affects the models W of $\mathcal{D}$.

However, in case of a complete postcondition, in order to be a capability model, we additionally require for a Web Service W that in the final state s_m of any execution $(s_0, \ldots, s_m)$ starting in a state s_0 where the precondition is satisfied, besides the postcondition, no other *positive* statement about the world (that is not entailed by the postcondition) holds, in other words, for every positive[8] formula $\psi \in \mathcal{L}(\Sigma)$ with $\phi^{post} \not\models_{\mathcal{L}} \psi$ the negation $\neg\psi$ must be satisfied in s_m.

Formally, this leads to the following extension of Definition 4.7 on capability satisfiability:

Definition 4.14 (Positive Formula). *A formula* $\psi \in \mathcal{L}(\Sigma)$ *is called* ***positive*** *it is not of the form* $\psi = \neg\psi'$ *for some* $\psi' \in \mathcal{L}(\Sigma)$*, i.e. if it does not use negation as the top-level operator.* □

Definition 4.15 (Capability Satisfaction, Capability Model in $\mathcal{F}_{comp}$). *Let* $W = (IF, \iota)$ *be a Web service in* $\mathcal{A}$ *and* $\mathcal{D} = (\phi^{pre}, (\phi^{post}, \gamma^{post})IF_{\mathcal{D}}) \in \mathcal{F}_{comp}$ *be a functional description of a Web service with complete and incomplete elements. Let* FV *denote the set of free variables occurring in* ϕ^{pre}, ϕ^{post} *and* ϕ^{inv} *and let* U *denote* $universe(\omega_{rw}(s_0))$.

W ***satisfies capability*** $\mathcal{D}$ *in* $\mathcal{A}$ *if and only if*

(i) *there exists a subset* $IF' \subseteq IF_{\mathcal{D}}$ *of the inputs of* $\mathcal{D}$ *and a bijection* $\pi : IF \to IF'$ *between* IF *and* IF' *such that*

(ii) *for all input bindings* $\beta \in In_{\mathcal{A}}(IF)$ *and abstract states* $s \in \mathcal{S}$*: for all* FV*-extensions* β' *of* $rename_{\pi}(\beta)$ *in* U*: if* $\iota(\beta, s) = (s_0, \ldots, s_m)$ *for some* $m \geq 0$ *and* $\omega_{rw}(s_0), \beta' \models_{\mathcal{L}(\Sigma)} \phi^{pre}$ *then* $\omega_{rw}(s_m), \beta' \models_{\mathcal{L}(\Sigma)} \phi^{post}$ *and*

(iii) *if* $\gamma^{post} = \texttt{complete}$ *then for all input bindings* $\beta \in In_{\mathcal{A}}(IF)$ *and abstract states* $s \in \mathcal{S}$*: for all* FV*-extensions* β' *of* $rename_{\pi}(\beta)$ *in* U*: if* $\iota(\beta, s) = (s_0, \ldots, s_m)$ *for some* $m \geq 0$ *and* $\omega_{rw}(s_0), \beta' \models_{\mathcal{L}(\Sigma)} \phi^{pre}$ *then* $\omega_{rw}(s_m), \beta' \models_{\mathcal{L}(\Sigma)} \neg\psi$ *for any positive formula* $\psi \in \mathcal{L}(\Sigma)$ *with* $\phi^{post} \not\models_{\mathcal{L}(\Sigma)} \psi$

In this case we write $W \models_{\mathcal{F}_{comp}} \mathcal{D}$ *and call the Web service* W *a* ***capability model*** *(or simply model) of* $\mathcal{D}$ *in* $\mathcal{A}$. □

Clauses (i) and (ii) are again precisely the same as in Definition 4.7 for our language $\mathcal{F}$. Only clause (iii) is added and captures the intended semantics of complete postconditions.

According to our new definition of satisfiability in $\mathcal{F}_{comp}$, we get the following relationship between $\mathcal{F}$ and $\mathcal{F}_{comp}$ which allows us to consider $\mathcal{F}$ as a special case

[8] We call a formula ψ positive if it does not use negation as the top-level operator, i.e. it is not of the form $\psi = \neg\psi'$ for some $\psi' \in \mathcal{L}(\Sigma)$

of $\mathcal{F}_{comp}$: For every description $\mathcal{D} = (\phi^{pre}, \phi^{post}, IF) \in \mathcal{F}$ there is a description $\mathcal{D}' \in \mathcal{F}_{comp}$ which is equivalent in any abstract state space $\mathcal{A}$, i.e. a description $\mathcal{D}'$ such that $W \models_{\mathcal{F}} \mathcal{D}$ iff. $W \models_{\mathcal{F}_{comp}} \mathcal{D}'$ for all Web Services W in $\mathcal{A}$ holds. In particular, $\mathcal{D}' = (\phi^{pre}, (\phi^{post}, \texttt{incomplete}), IF)$ has this property.

Extending the Semantic Analysis of Descriptions in $\mathcal{F}$. `to be done!`

Conclusion. We proposed an extension of our basic capability description language $\mathcal{F}$ that we consider as very useful in concrete applications, since it allows to easily express a meta-property of a description and prevents a modeler from tedious work. Furthermore, we expect that the case of complete descriptions occurs more often in practice than the case of incomplete descriptions, i.e. we expect Web service providers to give an optimistic perspective on what they provide. Hence, it would even make sense, to make the completeness marker optional and interpret the description element as being complete by default unless specified otherwise.

Remark 4.1 (Completeness for Executional Invariants). In the case of $\mathcal{F}_{inv}$, execution invariants are such candidates for descriptive elements of a capability which can be considered as incomplete or complete. However, for execution invariants, the requirement of completeness seems to be too strong to be useful, since according to the semantics of execution invariants this would imply that the state of the world would not be allowed to change at all in between the prestate and the finalstate, in other words, there can only be one intermediary state. At present, we do not consider this is useful and thus do not consider the distinction between complete and incomplete execution invariants – execution invariants are always considered as being incomplete. ◁

4.5 Limitation of the Model

The outlined model obviously has some limitations about what can be expressed or captured:

- **Only finite processes can be described.** A specific (inherent) limitation of pre- and postcondition style descriptions is that they are based on the assumption that there will be a final state of a computation, i.e. the computation terminates. Although, this might be a valid assumption for a wide variety of Web Services, it does not allow the specification of non-terminating components which deliver sensefull functionality though. An example in a technical system would be an operation system of a computer which does not terminate or a Web Service that implements a clock and periodically sends a the current time to a subscribed clients.

- **Statements about intermediate states.** Like in common specification frameworks, our model for the semantics of web services considers a web service as a set of *atomic* state-changes, i.e. possible intermediate states during an execution of a service are invisible for an external observer and can not be referred to in a formal specifications. For planing purposes it might be relevant or useful to allow to describe services in a more detailed way, for instance as a constraint on possible execution paths.

  For the same reason, it is not possible to express properties which do hold

during the whole execution of the service[9], which have been studied in Dynamic Logics in the context of *throughout* modalities. As an example think about the property of an account balance of a Please note, that the presented formal model can be adapted to this setting as well; this can essentially be achieved by replacing pairs (s, s') of states by sequences $(s_0, \ldots, s_n)$ of states in our model and the formal definitions. An assertion language $\mathcal{L}$ (used for expressing pre- and postconditions) which provides syntactic means to refer to intermediate states of a service execution can exploit this richer semantic structures. An example for a logic which provides a model-theoretic semantics of this kind as well as syntactic means for declaratively specifying sets of (possible) execution paths means to characterize is Transaction Logics [Bonner and Kifer, 1995].

For now, we stick to the following minimalistic principle: as long as there is no concrete evidence for the need of such expressivity, there is no need to generalize the model we presented above.

In this section we describe a proposal for applying the framework for functional description of web services which we outlined in the previous section to WSML.

- *keyword result is set of pre and postconditions*
- *way to define inputs*
- *@pre to reference predicates in pre state*
- *treat variable quantifiction*

We have defined Abstract State Spaces as a formal model for appropriately describing how Web services act in the world and change it. The main features of the proposed model are: (i) language independence to a maximum extent, and (ii) modular and flexible definitions that can easily be extended to fit the needs for specific applications.

Language independence, in particular, means that our approach is applicable to a variety of static description language (capturing properties of single states). Thus, it is especially suitable for application in frameworks like OWL-S and WSMO that describe the functionality provided by Web services in a state-based manner. On basis of our model, we have rigorously defined the semantics of functional descriptions. We demonstrated the applicability and benefit of our model in terms of a concrete use case, namely the semantic analysis of functional descriptions. Therein, we have illustrated how to capture several interesting and naturally arising properties of functional descriptions, in particular *functional refinement* and *realizability*. We have given mathematically concise definitions and exemplified how to device a provably correct algorithm for semantic analysis based on existing algorithms and systems. The use case followed throughout the explications supports our thesis: the correctness of any sort of symbolic computation based on functional descriptions of Web services can be analyzed and exposed in our framework.

While this paper presents the basic model, we plan to apply it to frameworks like WSMO and OWL-S that strive for genericity and independence of specific static languages for state descriptions. In particular, we plan to develop a matching mechanism following the defined notion of functional refinement in

[9]Such properties P are different from invariants, since they are guarantees local to a service rather than global properties of a system; Furthermore, they are different from strengthening pre- and postconditions by P since throughout-properties apply as well to any intermediate state of an execution.

order to provide a component with clear defined functionality for functional Web service discovery. Furthermore, we consider several extensions of the model, namely integrating *execution invariants* as properties that are guaranteed not to change during execution of a Web service (see [Lausen, 2005] for details), as well as integrating behavioral descriptions like choreography and orchestration interfaces that are concerned with the intermediate states in order to consume, respectively achieve the functionality of a Web service.

Bibliography

[Blackburn et al., 2001] Blackburn, P., de Rijke, M., and Venema, Y. (2001). *Modal Logic*. Cambridge University Press.

[Bonner and Kifer, 1995] Bonner, A. and Kifer, M. (1995). Transaction logic programming (or, a logic of procedural and declarative knowledge. Technical report.

[Bonner and Kifer, 1998a] Bonner, A. J. and Kifer, M. (1998a). A logic for programming database transactions. In *Logics for Databases and Information Systems*, pages 117–166.

[Bonner and Kifer, 1998b] Bonner, A. J. and Kifer, M. (1998b). Results on reasoning about updates in transaction logic. In *Transactions and Change in Logic Databases*, pages 166–196.

[Bonner and Kifer, 1998c] Bonner, A. J. and Kifer, M. (1998c). The state of change: A survey. In *Transactions and Change in Logic Databases*, pages 1–36.

[C.Green, 1969] C.Green (1969). Applications of theorem proving to problem solving. In *International Joint Conference on Artificial Intelligence (IJCAI 69)*, pages 219–240.

[Eiffel-Software, 2004] Eiffel-Software (2004). The Eiffel Homepage. `http://www.eiffel.com/`.

[Enderton, 2000] Enderton, H. B. (2000). *A Mathematical Introduction to Logic*. Academic Press, second edition edition.

[Fitting, 1996] Fitting, M. (1996). *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, second edition edition.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Hunter, 1998] Hunter, A. (1998). Paraconsistent logics. In Gabbay, D. and Smets, P., editors, *Handbook of Defeasible Reasoning and Uncertain Information*, volume 2. Kluwer. ISBN 0-7923-5161.

[Keller et al., 2005] Keller, U., Lara, R., Lausen, H., Polleres, A., and Fensel, D. (2005). Automatic Location of Services. In *Proceedings of 2nd European Semantic Web Conference (ESWC)*, pages 1–16.

[Keller and Lara (eds.),] Keller, U. and Lara (eds.), R. WSMO Web Service Discovery. Deliverable D5.1v0.1 Nov 12 2004, WSML Working Group. online: http://www.wsmo.org/TR/.

[Kifer et al., 2004] Kifer, M., Lara, R., Polleres, A., Zhao, C., Keller, U., Lausen, H., and Fensel, D. (2004). A logical framework for web service discovery. In *ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, volume 119, Hiroshima, Japan. CEUR Workshop Proceedings.

[Lausen, 2005] Lausen, H. (2005). Functional Description of Web Services. Deliverable D28.1v0.1 Oct 20 2005, WSML Working Group. online: http://www.wsmo.org/TR/.

[Lausen et al., 2005] Lausen, H., Polleres, A., and Roman (eds.), D. (2005). Web Service Modeling Ontology (WSMO). W3C Member Submission 3 June 2005. online: http://www.w3.org/Submission/WSMO/.

[Martin et al., 2005] Martin, D., McGuinness, D. L., Newton, G., DeRoure, D., Skall, M., and Yoshida, H. (2005). Semantic Web Services Framework (SWSF). W3C Member Submission 9 May 2005. online: http://www.w3.org/Submission/2005/07/.

[Martin (ed.), 2004] Martin (ed.), D. (2004). OWL-S: Semantic Markup for Web Services. W3C Member Submission 22 November 2004. online: http://www.w3.org/Submission/OWL-S.

[McCarthy, 1963] McCarthy, J. (1963). Situations, actions and causal laws. Technical report, Stanford University.

[Meyer, 1992] Meyer, B. (1992). *Eiffel: the Language*. Prentice Hall PTR.

[Meyer, 2000] Meyer, B. (2000). *Object-Oriented Software Construction*. Prentice Hall PTR, second edition edition.

[Object Management Group, 2005] Object Management Group (2005). The Unified Modelling Language Homepage. `http://www.uml.org/`.

[Paolucci et al., 2002] Paolucci, M., Kawamura, T., Payne, T. R., and Sycara, K. (2002). Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference*.

[Schmitt, 2001] Schmitt, P. H. (2001). Iterate logic. *Lecture Notes in Computer Science*, 2183:191–??

[Sycara et al., 1998] Sycara, K., Lu, J., and Klusch, M. (1998). Interoperability among heterogeneous software agents on the internet. Technical Report CMU-RI-TR-98-22, CMU, Pittsburgh, USA.

[Zaremski and Wing, 1997] Zaremski, A. M. and Wing, J. M. (1997). Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6(4):333–369.

A Proofs

In the following, we give formal proofs of the theorems which are mentioned in the document.

Theorem 4.1 Let $\mathcal{D}_1 = (\phi_1^{pre}, \phi_1^{post}, IF_1)$ and $\mathcal{D}_2 = (\phi_2^{pre}, \phi_2^{post}, IF_2)$ be functional descriptions in $\mathcal{F}$ with the same interfaces, i.e. $IF_1 = IF_2$. Let $[\phi]_{\Sigma_D^{pre} \to \Sigma_D}$ denote the formula ϕ' which can be derived from ϕ by replacing any dynamic symbol $\alpha \in \Sigma_D$ by its corresponding pre-variant $\alpha_{pre} \in \Sigma_D^{pre}$. Let $Cl_\forall(\phi)$ denote the universal closure of formula ϕ, i.e. a closed formula ϕ' of the form $\forall x_1, \ldots, x_n(\phi)$ where $x_1, \ldots, x_n$ are all the variables that occur free in ϕ. Let the entailment relation $\models_\mathcal{L}$ be defined in model-theoretic terms, i.e. $\phi_1 \models_\mathcal{L} \phi_2$ iff. for any interpretation I that satisfies ϕ_1 it holds I satisfies ϕ_2 too.

Then $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ if (i) $\Omega \models_\mathcal{L} Cl_\forall(\phi_2^{pre} \Rightarrow \phi_1^{pre})$ and
(ii) $\Omega \cup [\Omega]_{\Sigma_D^{pre} \to \Sigma_D} \models_\mathcal{L} Cl_\forall(([\phi_2^{pre}]_{\Sigma_D^{pre} \to \Sigma_D} \wedge \phi_1^{post}) \Rightarrow \phi_2^{post})$ □

Proof of Theorem 4.1 Let $\mathcal{A} = (\mathcal{S}, \mathcal{U}, \Sigma, \Omega, \omega)$ be an Abstract State Space. Let $\mathcal{D}_1 = (\phi_1^{pre}, \phi_1^{post}, IF_1)$ and $\mathcal{D}_2 = (\phi_2^{pre}, \phi_2^{post}, IF_2)$ be functional descriptions in $\mathcal{F}$. Assume (i) $\Omega \models_\mathcal{L} Cl_\forall(\phi_2^{pre} \Rightarrow \phi_1^{pre})$, (ii) $\Omega \cup [\Omega]_{\Sigma_D^{pre} \to \Sigma_D} \models_\mathcal{L} Cl_\forall(([\phi_2^{pre}]_{\Sigma_D^{pre} \to \Sigma_D} \wedge \phi_1^{post}) \Rightarrow \phi_2^{post})$ and (iii) $IF_1 = IF_2$. Let $W = (IF_W, \iota)$ be an arbitrary Web service in $\mathcal{A}$ with $W \models_\mathcal{F} \mathcal{D}_1$. If we can show that for this W it holds $W \models_\mathcal{F} \mathcal{D}_2$ as well, we have established $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$.

Following Def. 4.7, for W it holds that there exists a subset $IF_1' \subseteq IF_1$ of the inputs of $\mathcal{D}_1$ and a bijection $\pi_1 : IF_W \to IF_1'$ between IF_W and IF_1' such that for all input bindings $\beta \in In_\mathcal{A}(IF_W)$, all abstract states $s \in \mathcal{S}$ and all FV_1-extensions β' of $rename_{\pi_1}(\beta)$ in U: if $\iota(\beta, s) = (s_0, \ldots, s_m)$ for some $m \geq 0$ and $\omega_{rw}(s_0), \beta' \models_\mathcal{L} \phi_1^{pre}$ then $\omega_{rw}(s_m), \beta' \models_\mathcal{L} \phi_1^{post}$ $(*)$, where FV_1 denotes the set of free variables occurring in ϕ_1^{pre} and ϕ_1^{post} and $U = universe(\omega_{rw}(s_0))$

We now show $W \models_\mathcal{F} \mathcal{D}_2$: Consider $IF_2' := IF_1'$ and $\pi_2 := \pi_1$. Because of (iii) we have $IF_1 = IF_2$ and thus a subset $IF_2' \subseteq IF_2$ of the inputs of $\mathcal{D}_2$ and a bijection $\pi_2 : IF_W \to IF_2'$. Let β be an arbitrary input binding in $In_\mathcal{A}(IF_W)$ and $s \in \mathcal{S}$ be an arbitrary abstract state in $\mathcal{A}$. Consider the Web service execution $\iota(\beta, s) = (s_0, \ldots, s_m)$ and an arbitrary FV_2-extensions β_2' of $rename_{\pi_2}(\beta)$ in U, where FV_2 denotes the set of free variables occurring in ϕ_2^{pre} and ϕ_2^{post} and $U = universe(\omega_{rw}(s_0))$.

Assume that $\omega_{rw}(s_0), \beta_2' \models_\mathcal{L} \phi_2^{pre}$. We then need to show that $\omega_{rw}(s_m), \beta_2' \models_\mathcal{L} \phi_2^{post}$ holds as well. Def. 4.1 ensures that $\omega_{rw}(s_0) \models_\mathcal{L} \Omega$, hence (by the definition of logical entailment in $\mathcal{L}$) from (i) we can infer $\omega_{rw}(s_0) \models_\mathcal{L} Cl_\forall(\phi_2^{pre} \Rightarrow \phi_1^{pre})$ and eventually $\omega_{rw}(s_0), \beta_1' \models_\mathcal{L} \phi_1^{pre}$ for any FV_1-extension β_1' of β_2' in U. Clearly, β_1' is an FV_1-extension (in U) of $rename_{\pi_2}(\beta)$ and (because of $\pi_2 = \pi_1$) of $rename_{\pi_1}(\beta)$. Thus, we can apply $(*)$ and infer $\omega_{rw}(s_m), \beta_1' \models_\mathcal{L} \phi_1^{post}$ $(**)$. Futhermore, Def. 4.1 ensures that $\omega_{rw}(s_m) \models_\mathcal{L} \Omega$ as well as $\omega_{rw}(s_0) \models_\mathcal{L} \Omega$. Because of Clause (v) in Def. 4.3, we have for all $\alpha \in \Sigma_D$: $meaning_{\omega_{rw}(s_0)}(\alpha) = meaning_{\omega_{rw}(s_m)}(\alpha_{pre})$ and thus replacing dynamic symobls by their pre-variants in Ω does not affect the truth value of the modified ontology under the post-state interpretation, i.e. if $\omega_{rw}(s_0) \models_\mathcal{L} \Omega$ then $\omega_{rw}(s_m) \models_\mathcal{L} [\Omega]_{\Sigma_D^{pre} \to \Sigma_D}$ $(***)$. Hence we have shown $\omega_{rw}(s_m) \models_\mathcal{L} \Omega \cup [\Omega]_{\Sigma_D^{pre} \to \Sigma_D}$ and by applying (ii), we can conclude $\omega_{rw}(s_m) \models_\mathcal{L} Cl_\forall(([\phi_2^{pre}]_{\Sigma_D^{pre} \to \Sigma_D} \wedge \phi_1^{post}) \Rightarrow \phi_2^{post})$.

By our assumption $\omega_{rw}(s_0), \beta_2' \models_\mathcal{L} \phi_2^{pre}$ holds. Since β_2' is an FV_2-extension of β and $FreeVars(\phi_2^{pre}) \subseteq FV_2$, it assigns a value to every variable occurring

free in ϕ_2^{pre}. Extending β_2' by assignments of additional variables that do not occur in ϕ_2^{pre} (e.g. the modification to the FV_1-extension β_1') does not change the truth value of ϕ_2^{pre} under the same interpretation and thus $\omega_{rw}(s_0), \beta_1' \models_{\mathcal{L}} \phi_2^{pre}$ holds as well. By the same line of argument that we used above to show $(***)$, we can conclude that then $\omega_{rw}(s_m), \beta_1' \models_{\mathcal{L}} [\phi_2^{pre}]_{\Sigma_D^{pre} \to \Sigma_D}$ must hold.

Hence, by $(**)$ we have $\omega_{rw}(s_m), \beta_1' \models_{\mathcal{L}} [\phi_2^{pre}]_{\Sigma_D^{pre} \to \Sigma_D} \wedge \phi_1^{post}$. Eventually, we can apply again (ii) to conclude $\omega_{rw}(s_m), \beta_1' \models_{\mathcal{L}} \phi_2^{post}$. Since only the free variables occuring in ϕ_2^{post} are relevant for the truth value of ϕ_2^{post} under the variable binding β_1' and β_1' is an FV_1-extension of the FV_2-extension β_2' of β and $FreeVars(\phi_2^{post}) \subseteq FV_2$, we can infer from Def. 4.5 that $FreeVars(\phi_2^{post}) \subseteq dom(\beta_1')$ and $\beta_1'|_{FreeVars(\phi_2^{post})} = \beta_2'|_{FreeVars(\phi_2^{post})}$.

Consequently, $\omega_{rw}(s_m), \beta_2' \models_{\mathcal{L}} \phi_2^{post}$ must hold which completes the proof. □